AMIGA® ROM Kernel Reference Manual

LIBRARIES

AMIGA TECHNICAL REFERENCE SERIES

COMMODORE-AMIGA, INC.

THIRD EDITION

AMIGA®

ROM Kernel Reference Manual: Libraries

Third Edition

Commodore-Amiga, Inc.

AMIGA TECHNICAL REFERENCE SERIES

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Contributors:

Bruce Barrett, Mark Barton, Steve Beats, Dave Berezowski, Ray Brand, Bob Burns, Peter Cherna, Eric Cotton, Susan Deyl, Sam Dicker, Ken Farinsky, Stuart Ferguson, Andy Finkel, Chris Green, Darren Greenwald, Jerry Hartzler, Paul Higginbottom, Larry Hildenbrand, Randell Jesup, David Junod, Neil Katin, Joe Katz, Kevin Klop, Bill Koester, Adam Levin, Dave Lucas, Dale Luck, Chris Ludwig, Jim Mackraz, R.J. Mical, David Miller, Bryce Nesbitt, John Orr, Bob Pariseau, Rob Peck, Tom Pohorsky, Nancy Rains, Chris Raymond, Mark Ricci, Tom Rokicki, Carl Sassenrath, Stan Shepard, Jez San, Michael Sinz, Carolyn Scheppner, Leo Schwab, Spencer Shanson, Darius Taghavy, Martin Taillefer, Ewout Walraven, Bart Whitebook, John Wiederhirn and Rob Wyesham.

Third edition by:

Dan Baker

Cover designer:

Hannus Design Associates

2 3 4 5 6 7 8 9-KE-9695949392
Second printing, August 1992
ISBN 0-201-56774-1

WARNING: The information described in this manual may contain errors or bugs, and may not function as described. All information is subject to enhancement or upgrade for any reason including to fix bugs, add features, or change performance. As with all software upgrades, full compatibility, although a goal, cannot be guaranteed, and is in fact unlikely.

Contents

Introduction

User Interface Libraries

Exec Library

Graphics Libraries

Additional Libraries

Appendices

Preface

This edition of the *Amiga ROM Kernel Reference Manual: Libraries* provides the latest information on how to program the Amiga line of personal computers from Commodore. It has been updated for Release 2 of the Amiga operating system and covers the newest Amiga computer systems including the A3000.

This book is meant to help you learn how to program the Amiga. It assumes some previous experience with programming and familiarity with computers in general. Although it is not required, a knowledge of the C programming language will make it much easier to understand the material in this book. Most of the Amiga operating system is written in C (with the rest written in 68000 assembly language), hence C is the language used for the programming examples.

This book is intended for the following audiences:

- C and assembly language programmers who want to create application software for the Amiga line of personal computers.
- Amiga software developers who want to upgrade their software for Release 2 of the operating system.
- Anyone who wants to know more about how the Amiga system software works.

The Amiga system software is organized into related groups of functions called libraries. The same organization is used for this book. Here is a brief overview of the contents:

- Chapter 1, *Introduction to Amiga System Libraries*. A look at the Amiga software and hardware architecture with an introduction to the basic elements of Amiga programming.
- Chapters 2-16, *User Interface Libraries*. An in-depth tutorial on how to create a graphic user interface for Amiga application software using Intuition and related modules including GadTools, Workbench, BOOPSI and ASL.
- Chapters 17-26, *Exec Library*. The details on how Exec, the system's master module, controls the system with working examples of interrupt processing code, subtask creation, lists and queues, semaphores, message passing and signalling.
- Chapters 27-30, *Graphic Libraries*. A complete explanation of the functions in the graphic and layers library that drive the Amiga's display hardware with examples of text rendering, line drawing, animation and more.

- Chapters 31-37, *Additional Libraries.* Tutorials on how to use the Amiga commodities, DOS, IFFParse, keymap, translator and other important libraries in the operating system.

- *Appendices.* Special sections containing a debugging and troubleshooting guide plus a working example library for programmers who want to extend the capabilities of the operating system.

We suggest that you use this book according to your level of familiarity with the Amiga system. Beginners should read the first four chapters and try the examples to get the basics. Then browse through the Exec chapters to get a deeper understanding of how the system works.

Advanced Amiga programmers should read the chapters on new libraries like IFFParse and GadTools to find out what's new in Release 2. Also be sure to review the new Utility library to see how tag item lists have been used to implement many of the system improvements in Release 2.

There are four other manuals in the Amiga Technical Reference Series. The *Amiga ROM Kernel Reference Manual: Devices* is a companion book to this volume detailing how to write code for the Amiga's lower level I/O hardware. The *Amiga ROM Kernel Reference Manual: Includes and Autodocs* is an alphabetically organized reference of ROM function summaries and system include files. Both these books are required reading for the serious programmer.

Also available are the *Amiga User Interface Style Guide*, an application design specification and reference work describing how a standard Amiga application should look and feel; and the *Amiga Hardware Reference Manual*, an in-depth description of the custom chips and other hardware components underlying the Amiga's sophisticated design.

Chapter 1
INTRODUCTION TO AMIGA SYSTEM LIBRARIES

The Amiga, like other microcomputers, contains a ROM full of routines that make programming the machine easier. The purpose of this book is to show you how to use these routines. Perhaps the best way to learn Amiga programming is by following examples and that is the method used in this book. Before starting though it will be helpful to go over some Amiga fundamentals. This section presents some of the basics that all Amiga programmers need to know.

Programming in the Amiga Environment

To program in the Amiga's dynamic environment you need to understand these special features of the Amiga's design:

- Multitasking (without memory protection)
- Shared libraries of functions
- Dynamic memory architecture (no memory map)
- Operating system versions
- Custom chips with DMA access (two kinds of memory)

MULTITASKING

The key feature of the Amiga's operating system design is *multitasking*. Multitasking means many programs, or tasks, reside in memory at the same time sharing system resources with one another. Programs take turns running so it appears that many programs are running simultaneously.

Multitasking is based on the concept that a program spends most of its time waiting for things to happen. A program waits for events like key presses, mouse movement, or disk activity. While a program is waiting, the CPU is idle. The CPU could be used to run a different program during this idle period if there was a convenient method for rapidly switching from one program to another. This is what multitasking does.

What the System Does For You

The Amiga uses *preemptive multitasking* which means that the operating system keeps track of all the tasks in memory and decides which one should run. The system checks hundreds of times per second to see which task should be run based on whether or not it is waiting, and other factors. Since the system handles all the work of task switching, multitasking is transparent to the application. From the application's point of view, it appears to have the machine all to itself.

The Amiga OS also manages the sharing of resources between tasks. This is important because in order for a variety of tasks to run independently in the Amiga's multitasking environment, tasks must be prevented from interfering with one another. Imagine if five tasks were allowed to use the parallel port at the same time. The result would be I/O chaos. To prevent this, the operating system provides an arbitration method (usually a function call) for every system resource. For instance you must call a function, **AllocMem()**, to get exclusive access to a block of memory.

What the System Doesn't Do For You

The Amiga operating system handles most of the housekeeping needed for multitasking, but this does not mean that applications don't have to worry about multitasking at all. The current generation of Amiga systems do not have hardware memory protection, so there is nothing to stop a task from using memory it has not legally acquired. An errant task can easily corrupt some other task by accidentally overwriting its instructions or data. Amiga programmers need to be extra careful with memory; one bad memory pointer can cause the machine to crash (debugging utilities such as MungWall and Enforcer will prevent this).

In fact, Amiga programmers need to be careful with every system resource, not just memory. All system resources from audio channels to the floppy disk drives are shared among tasks. Before using a resource, you must ask the system for access to the resource. This may fail if the resource is already being used by another task.

Once you have control of a resource, no other task can use it, so give it up as soon as you are finished. When your program exits, you must give everything back whether it's memory, access to a file, or an I/O port. You are responsible for this, the system will not do it for you automatically.

> *What Every Amiga Programmer Should Know:* The Amiga is a *multitasking* computer. Keep in mind that other tasks are running at the same time as your application. Always ask the system for control of any resource you need; some other task may already be using it. Give it back as soon as you are done; another task may want to use it. This applies to just about every computing activity your application can perform.

LIBRARIES OF FUNCTIONS

Most of the routines that make up the Amiga's operating system are organized into groups called libraries. In order to call a function on the Amiga you must first open the library that contains the function. For example, if you want to call the **Read()** function to read data from disk you must first open the DOS library.

The system's master library, called Exec, is always open. Exec keeps track of all the other libraries and is in charge of opening and closing them. One Exec function, **OpenLibrary()**, is used to open all the other libraries.

Almost any program you write for the Amiga will have to call the **OpenLibrary()** function. Usage is as follows:

```
struct Library *LibBase;      /* Global: declare this above main() */

main()
{
LibBase = OpenLibrary("library.name",version);

if(!LibBase) { /* Library did not open, so exit       */ }
else         { /* Library opened, so use its functions */ }
}
```

`LibBase`

This is a pointer to the library structure in memory, often referred to as the *library base*. The library base must be global because the system uses it to handle the library's function calls. The name of this pointer is established by the system (you cannot use any name you want). Refer to the list below for the appropriate name.

`library.name`

This is a C string that describes the name of the library you wish to open. The list of Amiga library names is given below.

`version`

This should be set to the earliest acceptable library version. A value of 0 matches any version. A value of 33 means you require at least version 33, or a later version of the library. If the library version in the system is older than the one you specify, **OpenLibrary()** will fail (return 0).

The table listed on the next page shows all the function libraries that are currently part of the Amiga system software. Column one shows the name string to use with **OpenLibrary()**; column two shows the name of the global variable you should use to hold the pointer to the library; column three shows the oldest version of the library still in use.

Table 1-1: Parameters to Use With OpenLibrary()

Library Name (library.name)*	Library Base Name (LibBase)	Oldest Version In Use (version)
asl.library	AslBase	36
commodities.library	CxBase	36
diskfont.library	DiskfontBase	33
dos.library	DOSBase	33
exec.library	SysBase	33
expansion.library	ExpansionBase	33
gadtools.library	GadToolsBase	36
graphics.library	GfxBase	33
icon.library	IconBase	33
iffparse.library	IFFParseBase	36
intuition.library	IntuitionBase	33
keymap.library	KeymapBase	33
layers.library	LayersBase	33
mathffp.library	MathBase	33
mathtrans.library	MathTransBase	33
mathieeedoubbas.library	MathIeeeDoubBasBase	33
mathieeedoubtrans.library	MathIeeeDoubTransBase	33
mathieeesingbas.library	MathIeeeSingBasBase	33
mathieeesingtrans.library	MathIeeeSingTransBase	33
rexxsyslib.library	RexxSysBase	36
translator.library	TranslatorBase	33
utility.library	UtilityBase	36
workbench.library	WorkbenchBase	33

*Other libraries may exist that are not supplied by Commodore since it is a feature of the operating system to allow such libraries.

Opening a Library in C

Call **OpenLibrary()** to open an Amiga function library. **OpenLibrary()** returns the address of the library structure (or library base) which you must assign to a specific global system variable as specified in the table above (case is important).

If the library cannot open for some reason, the **OpenLibrary()** function returns zero. Here's a brief example showing how it's used in C.

```
/* easy.c: a complete example of how to open an Amiga function library in C.
 * In this case the function library is Intuition.  Once the Intuition
 * function library is open, any Intuition function can be called.  This
 * example uses the DisplayBeep() function of Intuition to flash the screen.
 * With SAS/C (Lattice), compile with lc -L easy.c
 */

/* Declare the return type of the functions we will use.              */
struct Library *OpenLibrary();      /* These Exec library functions can be   */
void           CloseLibrary();      /* called anytime (Exec is always open). */

void           DisplayBeep();       /* Before using this Intuition function, */
                                    /* the Intuition library must be opened  */
```

```
struct IntuitionBase *IntuitionBase; /* Get storage for the library base      */
                                     /* The base name MUST be IntuitionBase    */
int main()
{
    IntuitionBase=(struct IntuitionBase *)OpenLibrary("intuition.library",33L);

    if(IntuitionBase)                /* Check to see if it actually opened.  */
        {                            /* The Intuition library is now open so */
        DisplayBeep(0L);             /* any of its functions may be used.    */

        CloseLibrary(IntuitionBase); /* Always close a library if not in use. */
        }
    else                             /* The library did not open so return an */
       {                             /* error code.  The exit() function is   */
       exit(20);                     /* not part of the OS, it is part of the */
       }                             /* compiler link library.               */
}
```

Opening a Library in Assembler

Here's the same example written in 68000 assembler. The principles are the same as with C: you must always open a library before using any of its functions. However, in assembler, library bases are treated a little differently than in C. In C, you assign the library base you get from **OpenLibrary()** to a global variable and forget about it (the system handles the rest). In assembler, the library base must always be in register A6 whenever calling any of the functions in the library.

You get the library base for any library except Exec, by calling **OpenLibrary()**. For Exec, you get the library base from the longword in memory location 4 ($0000 0004). Exec is opened automatically by the system at boot time, and its library base is stored there.

```
**********************************************************************
* A complete ready-to-assemble example of how to open an Amiga function
* library in 68000 assembler.  In this case the Intuition function library
* is opened and one of its functions, DisplayBeep() is called.
*
* When calling an Amiga function, the library base pointer *must* be in
* A6 (the library is free to depend on this).  Registers D0, D1, A0
* and A1 may be destroyed by the library, all others will be preserved.
*
_AbsExecBase EQU  4          ;System pointer to Exec's library base

        XREF    _LVOOpenLibrary       ;Offset from Exec base for OpenLibrary()
        XREF    _LVOCloseLibrary      ;Offset from Exec base for CloseLibrary()
        XREF    _LVODisplayBeep       ;Offset from Intuition base for DisplayBeep()

        move.l  _AbsExecBase,a6         ;Move exec.library base to a6
        lea.l   IntuiName(pc),a1        ;Pointer to "intuition.library" string
        moveq   #33,d0                  ;Version of library needed
        jsr     _LVOOpenLibrary(a6)     ;Call Exec's OpenLibrary() and
        tst.l   d0                      ;check to see if it succeeded
        bne.s   open_ok
        moveq   #20,d0                  ;Set failure code
        rts                             ;Failed exit

open_ok move.l  d0,a6                   ;Put IntuitionBase in a6.
        suba.l  a0,a0                   ;Load zero into a0
        jsr     _LVODisplayBeep(a6)     ;Call Intuition's DisplayBeep()

        move.l  a6,a1                   ;Put IntuitionBase into a1
        move.l  _AbsExecBase,a6
        jsr     _LVOCloseLibrary(a6)    ;Call Exec's CloseLibrary()
        moveq   #0,d0                   ;Set return code
        rts

IntuiName:      dc.b 'intuition.library',0
        END
```

The Amiga library functions are set up to accept parameters in certain 68000 registers and always return results in data register D0. This allows programs and functions written in assembler to communicate quickly. It also eliminates the dependence on the stack frame conventions of any particular language.

Amiga library functions use registers D0, D1, A0 and A1 for work space and use register A6 to hold the library base. Do not expect these registers to be the same after calling a function. All routines return a full 32 bit longword unless noted otherwise.

Another Kind of Function Library

The Amiga has two kinds of libraries: run-time libraries and link libraries. All the libraries discussed so far are run-time libraries. Run-time libraries make up most of the Amiga's operating system and are the main topic of this book.

There is another type of library known as a link library. Even though a link library is a collection of functions just like a run-time library, there are some major differences in the two types.

Run-time libraries

A run-time, or shared library is a group of functions managed by Exec that resides either in ROM or on disk (in the LIBS: directory). A run-time library must be opened before it can be used (as explained above). The functions in a run-time library are accessed dynamically at run-time and can be used by many programs at once even though only one copy of the library is in memory. A disk based run-time library is loaded into memory only if requested by a program and can be automatically flushed from memory when no longer needed.

Link libraries

A link library is a group of functions on disk that are managed by the compiler at link time. Link libraries do not have to be opened before they are used, instead you must link your code with the library when you compile a program. The functions in a link library are actually copied into every program that uses them. For instance the **exit()** function used in the C program listed above is not part of any of the libraries that make up the Amiga OS. It comes from the link library supplied with the compiler (*lc.lib* for SAS/Lattice C or *c.lib* for Manx Aztec C). The code that performs the **exit()** function is copied into the program when it is compiled.

Libraries, Devices and Resources

Most of the Amiga's OS routines are organized into groups of shared run-time libraries. The Amiga also has specialized function groups called *devices* and *resources* that programmers use to perform basic I/O operations or access low-level hardware.

Devices and resources are similar in concept to a shared run-time library. They are managed by Exec and must be opened before they can be used. Their functions are separate from the programs that use them and are accessed dynamically at run time. Multiple programs can access the device or resource even though only one copy exists in memory (a few resources can only be used by one program at a time.)

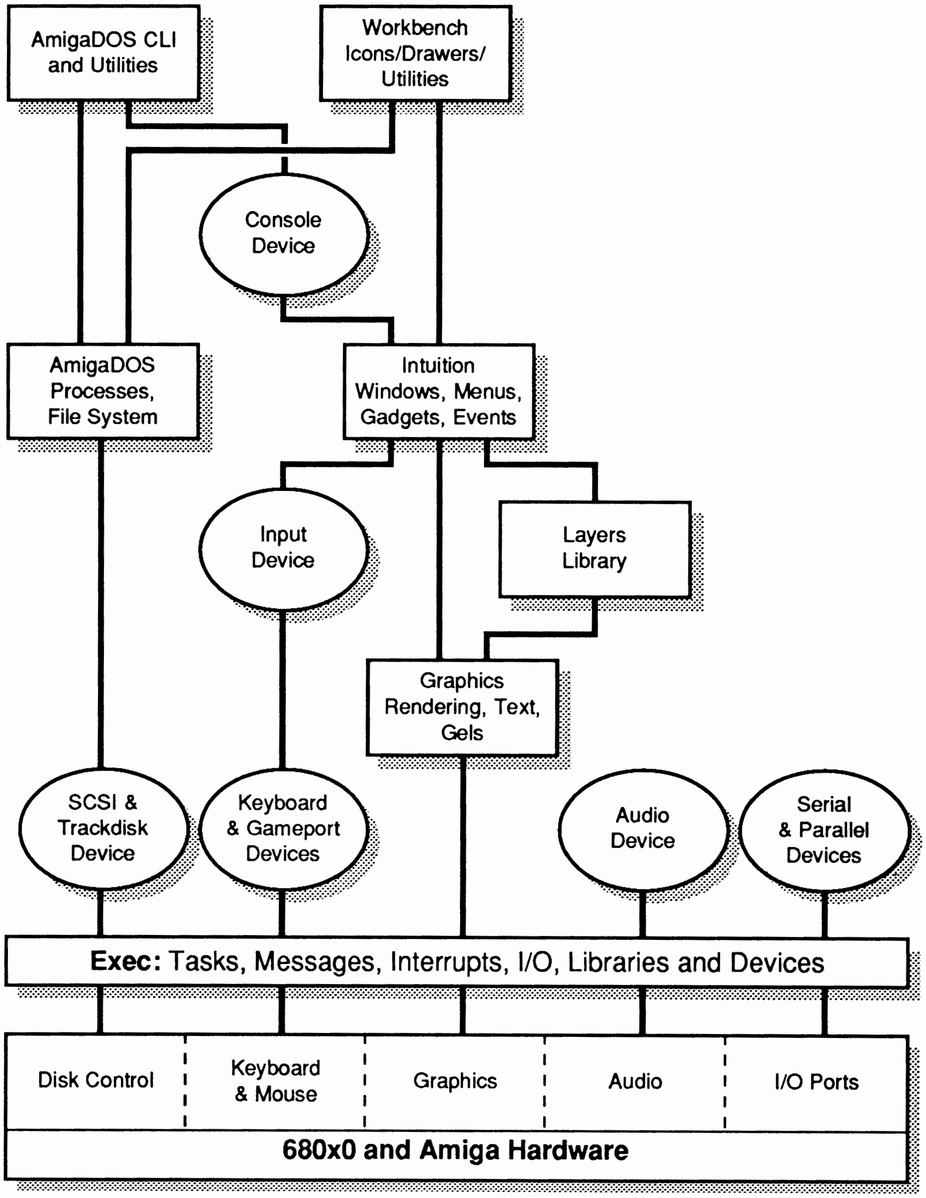


Figure 1-1: Amiga System Software Hierarchy

Devices and resources are managed by Exec just as libraries are. For more information on devices and resources, see the chapter on "Exec Device I/O" later in this book or refer to the *Amiga ROM Kernel Reference Manual: Devices* for detailed descriptions of each device.

> *What Every Amiga Programmer Should Know:* The functions in the Amiga OS are accessed through shared run-time libraries. Libraries must be opened before their functions may be used. The system's master library, Exec, is always open. The Exec function **OpenLibrary()** is used to open all other libraries.

DYNAMIC MEMORY ARCHITECTURE

Unlike some microcomputer operating systems, the Amiga OS relies on absolute memory addresses as little as possible. Instead the Amiga OS uses a technique (sometimes referred to as soft machine architecture) which allows system routines and data structures to be positioned anywhere in memory.

Amiga run-time libraries may be positioned anywhere in memory because they are always accessed through a jump table. Each library whether in ROM or loaded from disk has an associated **Library** structure and jump table in RAM.

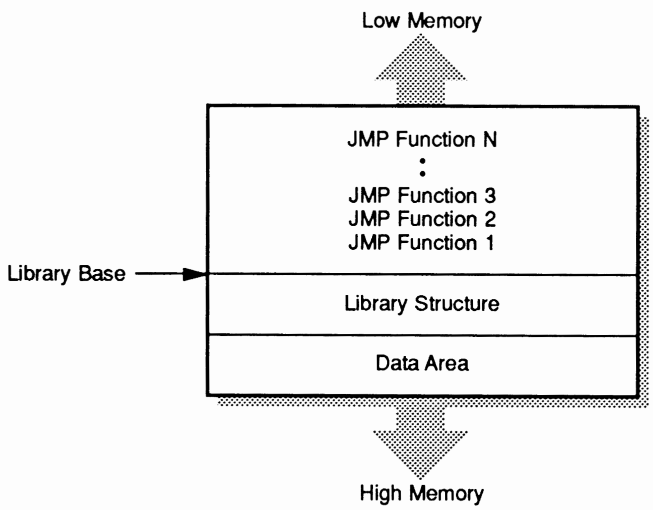


Figure 1-1: Amiga Library Structure and Jump Table

The system knows where the jump table starts in RAM because when a library is opened for the first time, Exec creates the library structure and keeps track of its location. The order of the entries in the library's jump table is always preserved between versions of the OS but the functions they point to can be anywhere in memory. Hence, system routines in ROM may be moved from one version of the OS to another. Given the location of the jump table and the appropriate offset into the table, any function can always be found.

Not only are system routines relocatable but system data structures are too. In the Amiga's multitasking environment, multiple applications run at the same time and each may have its own screen, memory, open files, and even its own subtasks. Since any number of application tasks are run and stopped at the user's option, system data structures have to be set up as needed. They cannot be set up ahead of time at a fixed memory location because there is no way to tell how many and what type will be needed.

The Amiga system software manages this confusion by using *linked lists* of information about items such as libraries, tasks, screens, files and available memory. A linked list is a chain of data items with each data item containing a pointer to the next item in the chain. Given a pointer to the first item in a linked list, pointers to all the other items in the chain can be found.

Exec: The System Executive

On the Amiga, the module that keeps track of linked lists is Exec, the system executive. Exec is the heart of the Amiga operating system since it also is in charge of multitasking, granting access to system resources (like memory) and managing the Amiga library system.

As previously discussed, memory location 4 ($0000 0004), also known as SysBase, contains a pointer to the Exec library structure. This is the only absolutely defined location in the Amiga operating system. A program need only know where to find the Exec library to find, use and manipulate all other system code and data.

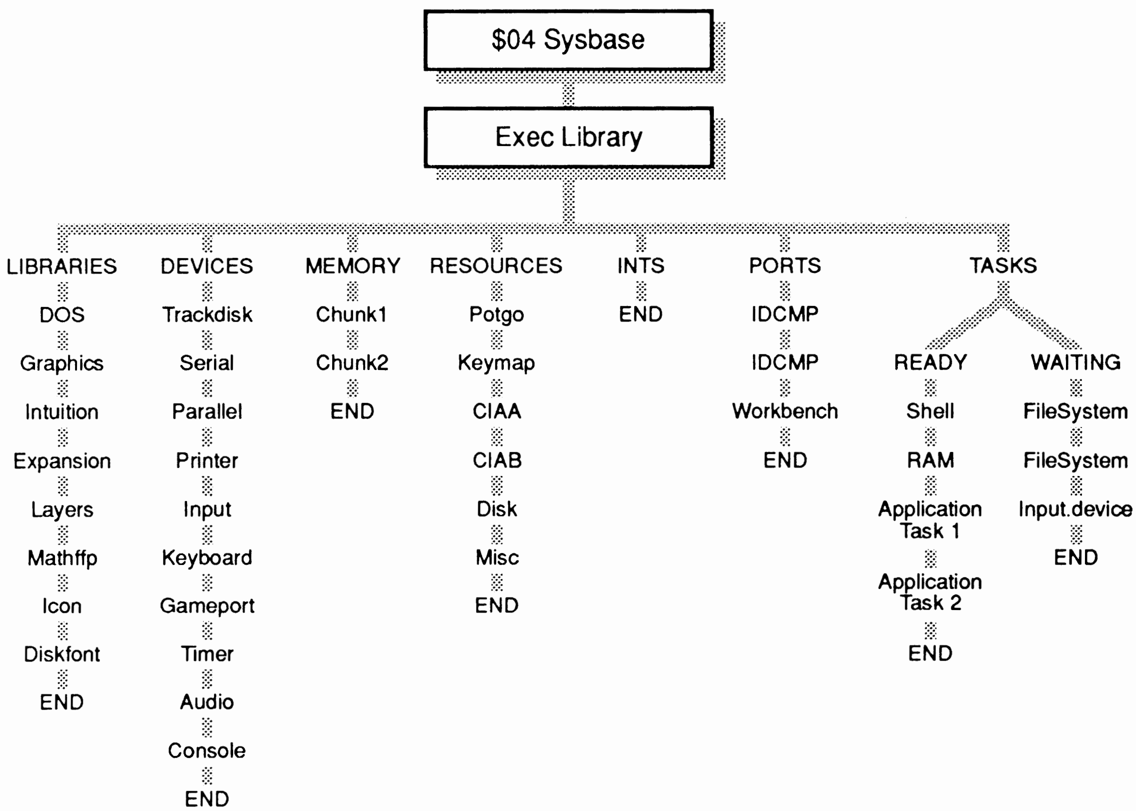


Figure 1-2: Exec and the Organization of the Amiga OS

The diagram above shows how the entire Amiga operating system is built as a tree starting at SysBase. Exec keeps linked lists of all the system libraries, devices, memory, tasks and other data structures. Each of these in turn can have its own variables and linked lists of data structures built onto it. In this way, the flexibility of the OS is preserved so that upgrades can be made without jeopardizing compatibility.

> *What Every Amiga Programmer Should Know:* The Amiga has a dynamic memory map. There are no fixed locations for operating system variables and routines. Do not call ROM routines or access system data structures directly. Instead use the indirect access methods provided by the system.

OPERATING SYSTEM VERSIONS

The Amiga operating system has undergone several major revisions summarized in the table below. The latest revision is Release 2 (corresponds to library versions 36 and above).

System library version number	Kickstart release
0	Any version
30	Kickstart V1.0 (obsolete)
31	Kickstart V1.1 (NTSC only - obsolete)
32	Kickstart V1.1 (PAL only - obsolete)
33	Kickstart V1.2 (the oldest revision still in use)
34	Kickstart V1.3 (adds autoboot to V33)
35	Special Kickstart version to support A2024 high-resolution monitor
36	Kickstart V2.0 (old version of Release 2)
37	Kickstart V2.04 (current version of Release 2)

The examples listed throughout this book assume you are using Release 2.

Many of the libraries and functions documented in this manual are available in all versions of the Amiga operating system. Others are completely new and cannot be used unless you have successfully opened the appropriate version of the library.

To find out which functions are new with Release 2 refer to the *ROM Kernel Reference Manual: Includes and Autodocs*. The functions which are new are marked with *(V36)* or *(V37)* in the NAME line of the function Autodoc. These new functions require you to use a matching version number (36, 37, or higher) when opening the library.

Exit gracefully and informatively if the required library version is not available.

About Release 2

Release 2 first appeared on the Amiga 3000. This initial version corresponds to Kickstart V2.00, system library version number V36. Release 2 was subsequently revised and this older version is now considered obsolete.

Programs written for Release 2 should use only the later version corresponding to Kickstart V2.04, system library version number V37. If your system is using the earlier version of Release 2, you should upgrade your system. (Upgrade kits may be obtained from an authorized Commodore service center.)

> *What Every Amiga Programmer Should Know:* Some libraries or specific functions are not available in older versions of the Amiga operating system. Be sure to ask for the lowest library version that meets the requirements of your program.

THE CUSTOM CHIPS

The most important feature of the Amiga's hardware design is the set of custom chips that perform specialized tasks independently of the CPU. Each of the custom chips (named Paula, Agnus, and Denise) is dedicated to a particular job:

Paula	(8364)	Audio, floppy disk, serial, interrupts
Agnus	(8361/8370/8372)	Copper (video coprocessor), blitter, DMA control
Denise	(8362)	Color registers, color DACs (Digital to Analog Converters) and sprites

The custom chips can perform work independently of the CPU because they have *DMA*, or *Direct Memory Access*, capability. DMA means the custom chips can access special areas of memory by themselves without any CPU involvement. (On computer systems without DMA, the CPU must do some or all of the memory handling for support chips.) The Amiga's custom chips make multitasking especially effective because they can handle things like rendering graphics and playing sound independently, giving the CPU more time to handle the overhead of task-switching and other important jobs.

Custom Chip Revisions

The custom chips have been revised as the Amiga platform has evolved and newer models of the Amiga developed. The latest revision of the Amiga custom chips is known as the *Enhanced Chip Set*, or ECS. Certain features of the Amiga operating system, such as higher resolution screens and special genlock modes, require the ECS version of the custom chips. In this book, features that require ECS are noted in the accompanying text. For more details about the special features of ECS, see Appendix C of the *Amiga Hardware Reference Manual.*

Two Kinds of Memory

To keep the Amiga running efficiently, the Amiga has two memory buses and two kinds of memory. *Chip memory* is memory that both the CPU and custom chips can access. *Fast memory* is memory that only the CPU (and certain expansion cards) can access. Since Chip memory is shared, CPU access may be slowed down if the custom chips are doing heavy-duty processing. CPU access to Fast memory is never slowed down by contention with the custom chips.

The distinction between Chip memory and Fast memory is very important for Amiga programmers to keep in mind because any data accessed directly by the custom chips such as video display data, audio data or sprite data *must be in Chip memory.*

> *What Every Amiga Programmer Should Know:* The Amiga has *two* kinds of memory: *Chip* memory and *Fast* memory. Use the right kind.

About the Examples

For the most part, the examples in this book are written in C (there are a few 68000 assembly language examples too).

C examples have been compiled under SAS C, version 5.10a. The compiler options used with each example are noted in the comments preceding the code.

In general, the examples are also compatible with Manx Aztec C 68K, version 5.0d, and other C compilers, however some changes will usually be necessary. Specifically, all the C examples assume that the automatic Ctrl-C feature of the compiler has been disabled. For SAS C (and Lattice C revisions 4.0 and greater) this is handled with:

```
/* Add this before main() to override the default Ctrl-C handling
 * provided in SAS (Lattice) C.  Ctrl-C event will be ignored */

int CXBRK ( void )    { return(0); }
int chkabort( void ) { return(0); }
```

For Manx Aztec C, replace the above with:

```
/* Add this near the top */
     #include <functions.h>

/* Add this before main() */
     extern int Enable_Abort;    /* reference abort enable */

/* Add this after main(), as the first active line in the program */
     Enable_Abort=0;         /* turn off CTRL-C */
```

Other changes may be required depending on the example and the C compiler you are using. Most of the C examples in this book use the following special option flags of the SAS/C compiler (set the equivalent option of whatever compiler you are using):

```
-b1 = Small data model.
-cf = Check for function prototypes.
  i = Ignore #include statements that are identical to one already given.
  s = Store all literal strings that are identical in the same place.
  t = Enable warnings for structures that are used before they are defined.

 -v = Do not include stack checking code with each function.
 -y = Load register A4 with the data section base address on function entry.
      The -v and -y flags are are generally only needed for parts of the
      program that are called directly by the system such as interrupt
      servers, subtasks, handlers and callback hook functions.
```

Except where noted, each example was linked with the standard SAS/C startup code *c.o*, the SAS/C linker library *lc.lib* and the Commodore linker library *amiga.lib*. The SAS/C compiler defaults to 32-bit ints. If your development environment uses 16-bit ints you may need to explicitly cast certain arguments as longs (for example `1L << sigbit` instead of `1 << sigbit`).

The 68000 assembly language examples have been assembled under the Innovatronics CAPE assembler V2.x, the HiSoft Devpac assembler V1.2, and the Lake Forest Logic ADAPT assembler 1.0. No substantial changes should be required to switch between assemblers.

General Amiga Development Guidelines

In the earlier sections of this chapter, the basic environment of the Amiga operating system was discussed. This sections presents specific guidelines that all Amiga programmers must follow. Some of these guidelines are for advanced programmers or apply only to code written in assembly language.

- Check for memory loss. Arrange your Workbench screen so that you have a Shell available and can start your program without rearranging any windows. In the Shell window type Avail flush several times (the flush option requires the Release 2 version of the Avail command). Note the total amount of free memory. Run your program (do not rearrange any windows other than those created by the program) and then exit. At the Shell, type Avail flush several times again. Compare the total amount of free memory with the earlier figure. They should be the same. Any difference indicates that your application is not freeing some memory it used or is not closing a disk-loaded library, device or font it opened. Note that under Release 2, a small amount of memory loss is normal if your application is the first to use the audio or narrator device.

- Use all of the program debugging and stress tools that are available when writing and testing your code. New debugging tools such as Enforcer, MungWall, and Scratch can help find uninitialized pointers, attempted use of freed memory and misuse of scratch registers or condition codes (even in programs that appear to work perfectly).

- Always make sure you actually get any system resource that you ask for. This applies to memory, windows, screens, file handles, libraries, devices, ports, etc. Where an error value or return is possible, ensure that there is a reasonable failure path. Many poorly written programs will appear to be reliable, until some error condition (such as memory full or a disk problem) causes the program to continue with an invalid or null pointer, or branch to untested error handling code.

- Always clean up after yourself. This applies for both normal program exit and program termination due to error conditions. Anything that was opened must be closed, anything allocated must be deallocated. It is generally correct to do closes and deallocations in reverse order of the opens and allocations. Be sure to check your development language manual and startup code; some items may be closed or deallocated automatically for you, especially in abort conditions. If you write in the C language, make sure your code handles Ctrl-C properly.

- Remember that memory, peripheral configurations, and ROMs differ between models and between individual systems. Do not make assumptions about memory address ranges, storage device names, or the locations of system structures or code. Never call ROM routines directly. Beware of any example code you find that calls routines at addresses in the $F0 0000 - $FF FFFF range. These are ROM routines and they will move with every OS release. The only supported interface to system ROM code is through the library, device, and resource calls.

- Never assume library bases or structures will exist at any particular memory location. The only absolute address in the system is $0000 0004, which contains a pointer to the Exec library base. Do not modify or depend on the format of private system structures. This includes the poking of copper lists, memory lists, and library bases.

- Never assume that programs can access hardware resources directly. Most hardware is controlled by system software that will not respond well to interference from other programs. Shared hardware requires programs to use the proper sharing protocols. Use the defined interface; it is the best way to ensure that your software will continue to operate on future models of the Amiga.

- Never access shared data structures directly without the proper mutual exclusion (locking). Remember that other tasks may be accessing the same structures.

- The system does not monitor the size of a program's stack. (Your compiler may have an option to do this for you.) Take care that your program does not cause stack overflow and provide extra stack space for the possibility that some functions may use up additional stack space in future versions of the OS.

- Never use a polling loop to test signal bits. If your program waits for external events like menu selection or keystrokes, do not bog down the multitasking system by busy-waiting in a loop. Instead, let your task go to sleep by **Wait()**ing on its signal bits. For example:

```
signals = (ULONG)Wait(  (1<<windowPtr->UserPort->mp_SigBit) |
                        (1<<consoleMsgPortPtr->mp_SigBit)  );
```

  This turns the signal bit number for each port into a mask, then combines them as the argument for the Exec library **Wait()** function. When your task wakes up, handle all of the messages at each port where the **mp_SigBit** is set. There may be more than one message per port, or no messages at the port. Make sure that you **ReplyMsg()** to all messages that are not replies themselves. If you have no signal bits to **Wait()** on, use **Delay()** or **WaitTOF()** to provide a measured delay.

- Tasks (and processes) execute in 680x0 user mode. Supervisor mode is reserved for interrupts, traps, and task dispatching. Take extreme care if your code executes in supervisor mode. Exceptions while in supervisor mode are deadly.

- Most system functions require a particular execution environment. All DOS functions and any functions that might call DOS (such as the opening of a disk-resident library, font, or device) can only be executed from a process. A task is not sufficient. Most other ROM kernel functions may be executed from tasks. Only a few may be executed from interrupts.

- Never disable interrupts or multitasking for long periods. If you use **Forbid()** or **Disable()**, you should be aware that execution of any system function that performs the **Wait()** function will temporarily suspend the **Forbid()** or **Disable()** state, and allow multitasking and interrupts to occur. Such functions include almost all forms of DOS and device I/O, including common *stdio* functions like **printf()**.

- Never tie up system resources unless it is absolutely necessary. For example, if your program does not require constant use of the printer, open the printer device only when you need it. This will allow other tasks to use the printer while your program is running. You must provide a reasonable error response if a resource is not available when you need it.

- All data for the custom chips must reside in Chip memory (type MEMF_CHIP). This includes bitplanes, sound samples, trackdisk buffers, and images for sprites, bobs, pointers, and gadgets. The **AllocMem()** call takes a flag for specifying the type of memory. A program that specifies the wrong type of memory may appear to run correctly because many Amigas have only Chip memory. (On all models of the Amiga, the first 512K of memory is Chip memory. In later models, Chip memory may occupy up to the first one or two megabytes).

  However, once expansion memory has been added to an Amiga (type MEMF_FAST), any memory allocations will be made in the expansion memory area by default. Hence, a program can run correctly on an unexpanded Amiga which has only Chip memory while crashing on an Amiga which has expanded memory. A developer with only Chip memory may fail to notice that memory was incorrectly specified.

Most compilers have options to mark specific data structures or object modules so that they will load into Chip RAM. Some older compilers provide the Atom utility for marking object modules. If this method is unacceptable, use the **AllocMem()** call to dynamically allocate Chip memory, and copy your data there.

When making allocations that do not require Chip memory, do not explicitly ask for Fast memory. Instead ask for memory type MEMF_PUBLIC or 0L as appropriate. If Fast memory is available, you will get it.

- Never use software delay loops! Under the multitasking operating system, the time spent in a loop can be better used by other tasks. Even ignoring the effect it has on multitasking, timing loops are inaccurate and will wait different amounts of time depending on the specific model of Amiga computer. The timer device provides precision timing for use under the multitasking system and it works the same on all models of the Amiga. The AmigaDOS **Delay()** function or the graphics library **WaitTOF()** function provide a simple interface for longer delays. The 8520 I/O chips provide timers for developers who are bypassing the operating system (see the *Amiga Hardware Reference Manual* for more information).

- Always obey structure conventions!

  - All non-byte fields must be word-aligned. Longwords should be longword-aligned for performance.
  - All address pointers should be 32 bits (not 24 bits). Never use the upper byte for data.
  - Fields that are not defined to contain particular initial values must be initialized to zero. This includes pointer fields.
  - All reserved or unused fields must be initialized to zero for future compatibility.
  - Data structures to be accessed by the custom chips, public data structures (such as a task control block), and structures which must be longword aligned must *not* be allocated on a program's stack.
  - Dynamic allocation of structures with **AllocMem()** provides longword aligned memory of a specified type with optional initialization to zero, which is useful in the allocation of structures.

FOR 68010/68020/68030/68040 COMPATIBILITY

Special care must be taken to be compatible with the entire family of 68000 processors:

- Do not use the upper 8 bits of a pointer for storing unrelated information. The 68020, 68030, and 68040 use all 32 bits for addressing.

- Do not use signed variables or signed math for addresses.

- Do not use software delay loops, and do not make assumptions about the order in which asynchronous tasks will finish.

- The stack frame used for exceptions is different on each member of the 68000 family. The type identification in the frame must be checked! In addition, the interrupt autovectors may reside in a different location on processors with a VBR register.

- Do not use the `MOVE SR,<dest>` instruction! This 68000 instruction acts differently on other members of the 68000 family. If you want to get a copy of the processor condition codes, use the Exec library **GetCC()** function.

- Do not use the `CLR` instruction on a hardware register which is triggered by Write access. The 68020 `CLR` instruction does a single Write access. The 68000 `CLR` instruction does a Read access first, then a Write access. This can cause a hardware register to be triggered twice. Use `MOVE.x #0, <address>` instead.

- Self-modifying code is strongly discouraged. All 68000 family processors have a pre-fetch feature. This means the CPU loads instructions ahead of the current program counter. Hence, if your code modifies or decrypts itself just ahead of the program counter, the pre-fetched instructions may not match the modified instructions. The more advanced processors prefetch more words. If self-modifying code must be used, flushing the cache is the safest way to prevent troubles.

- The 68020, 68030 and 68040 processors all have instruction caches. These caches store recently used instructions, but do not monitor writes. After modifying or directly loading instructions, the cache must be flushed. See the Exec library **CacheClearU()** Autodoc for more details. If your code takes over the machine, flushing the cache will be trickier. You can account for the current processors, and hope the same techniques will work in the future:

```
CACRF_ClearI      EQU   $0008       ;Bit for clear instruction cache
;
;Supervisor mode only. Use this only if you have taken over the
;machine.  Read and store the ExecBase processor AttnFlags flags at
;boot time, call this code only if the "68020 or better" bit was set.
;
ClearICache:     dc.w    $4E7A,$0002      ;MOVEC CACR,D0
                 tst.w   d0               ;movec does not affect CC's
                 bmi.s   cic_040          ;A 68040 with enabled cache!
                 ori.w   #CACRF_ClearI,d0
                 dc.w    $4E7B,$0002      ;MOVEC D0,CACR
                 bra.s   cic_exit
cic_040:         dc.w    $f4b8            ;CPUSHA (IC)
cic_exit:
```

HARDWARE PROGRAMMING GUIDELINES

If you find it necessary to program the hardware directly, then it is your responsibility to write code that will work correctly on the various models and configurations of the Amiga. Be sure to properly request and gain control of the hardware resources you are manipulating, and be especially careful in the following areas:

- Kickstart 2.0 uses the 8520 Complex Interface Adaptor (CIA) chips differently than 1.3 did. To ensure compatibility, you must always ask for CIA access using the cia.resource **AddICRVector()** and **RemICRVector()** functions. Do not make assumptions about what the system might be using the CIA chips for. If you write directly to the CIA chip registers, do not expect system services such as the trackdisk device to function. If you are leaving the system up, do not read or write to the CIA Interrupt Control Registers directly; use the cia.resource **AbleICR()**, and **SetICR()** functions. Even if you are taking over the machine, do not assume the initial contents of any of the CIA registers or the state of any enabled interrupts.

- All custom chip registers are Read-only or Write-only. Do not read Write-only registers, and do not write to Read-only registers.
- Never write data to, or interpret data from the unused bits or addresses in the custom chip space. To be software-compatible with future chip revisions, all undefined bits must be set to zeros on writes, and must be masked out on reads before interpreting the contents of the register.
- Never write past the current end of custom chip space. Custom chips may be extended or enhanced to provide additional registers, or to use bits that are currently undefined in existing registers.
- Never read, write, or use any currently undefined address ranges or registers. The current and future usage of such areas is reserved by Commodore and is subject to change.
- Never assume that a hardware register will be initialized to any particular value. Different versions of the OS may leave registers set to different values. Check the *Amiga Hardware Reference Manual* to ensure that you are setting up all the registers that affect your code.

ADDITIONAL ASSEMBLER DEVELOPMENT GUIDELINES

If you are writing in assembly language there are some extra rules to keep in mind in addition to those listed above.

- Never use the TAS instruction on the Amiga. System DMA can conflict with this instruction's special indivisible read-modify-write cycle.
- System functions must be called with register A6 containing the library or device base. Libraries and devices assume A6 is valid at the time of any function call. Even if a particular function does not currently require its base register, you must provide it for compatibility with future system software releases.
- Except as noted, system library functions use registers D0, D1, A0, and A1 as scratch registers and you must consider their former contents to be lost after a system library call. The contents of all other registers will be preserved. System functions that provide a result will return the result in D0.
- Never depend on processor condition codes after a system call. The caller must test the returned value before acting on a condition code. This is usually done with a TST or MOVE instruction.

1.3 Compatibility Issues

This 3rd edition of the *Amiga Technical Reference Series* focuses on the Release 2 version of the Amiga operating system (Kickstart V2.04, V37). Release 2 of the operating system was first shipped on the Amiga 3000 and now available as an upgrade kit for the Amiga 500 and Amiga 2000 models to replace the older 1.3 (V34) operating system. Release 2 contains several new libraries and hundreds of new library functions and features to assist application writers.

DESIGN DECISIONS

The latest Amiga models, including all A3000's, are running Release 2. But many older Amigas are still running 1.3 at this time. Depending on your application and your market, you may choose to require the Release 2 operating system as a minimum platform. This can be a reasonable requirement for vertical markets and professional applications. Release 2 can also be a reasonable requirement for new revisions of existing software products, since you could continue to ship the older 1.3-compatible release in the same package. If you have made the decision to require Release 2, then you are free to take advantage of all of the new libraries and features that Release 2 provides.

Throughout this latest edition of the *Amiga Technical Reference Series*, features, functions and libraries that are new for Release 2 are usually indicated by *(V36)* or *(V37)* in the description of the feature. Such features are not available on Amiga models that are running 1.3 (V34) or earlier versions of the OS. Unconditional use of Release 2 functions will cause a program to fail when it is run on a machine with the 1.3 OS. It is *very* important to remember this when designing and writing your code.

Developers of consumer-priced productivity, entertainment and utility software may not yet be ready to write applications that *require* Release 2, but even these developers can enhance their products by *taking advantage of Release 2 while remaining 1.3 compatible.*

There are three basic methods that will allow you to take advantage of enhanced Release 2 features while remaining 1.3 compatible:

- Transparent Release 2 Extensions
- Conditional Code
- Compatible Libraries

Transparent Release 2 Extensions

To provide Release 2 enhancements while remaining compatible with the older 1.3 version of the OS, several familiar 1.3 system structures have been extended to include an optional pointer to additional information. The new extended versions of such structures are generally defined in the same include file as the original structure. These extended structures are passed to the same 1.3 system functions as the unextended structure (e.g., **OpenWindow()**, **OpenScreen**, **AddGadget**, **OpenDiskFont()**). The existence of the extended information is signified by setting a new flag bit in the structure. (In one case, PROPNEWLOOK, only the flag bit itself is significant). These extensions are *transparent* to previous versions of the operating system. *Only* the Release 2 operating system will recognize the bit and act on the extended information.

The table below lists the flag bit for each structure to specify that extended information is present.

Original	Extended	Flag Field	Flag Bit	Defined In
NewScreen	ExtNewScreen	Type	NS_EXTENDED	*<intuition/screens.h>*
NewWindow	ExtNewWindow	Flags	WFLG_NW_EXTENDED	*<intuition/intuition.h>*
Gadget	Gadget	Flags	GFLG_STRINGEXTEND	*<intuition/intuition.h>*
PropInfo	PropInfo	Flags	PROPNEWLOOK	*<intuition/intuition.h*
TextAttr	TTextAttr	tta_Style	FSF_TAGGED	*<graphics/text.h>*

Through the use of such extensions, applications can request special Release 2 features in a 1.3-compatible manner. When the application is run on a Release 2 machine, the enhanced capabilities will be active.

The enhancements available through these extensions include:

Screen: Overscan, 3D Look (SA_Pens), public screens, PAL/NTSC, new modes
Window: Autoadjust sizing, inner dimensions, menu help
Gadget: Control of font, pens, and editing of string gadgets
PropInfo: Get 3D Look proportional gadgets when running under Release 2
TTextAttr: Control width of scaled fonts

Extensible longword arrays called **TagItem** lists are used to specify the extended information for many of these structures. Tag lists provide an open-ended and backwards-compatible method of growing system structures by storing all new specifications in an extendible array of longwords pairs.

Another transparent Release 2 extension is the diskfont library's ability to scale bitmap and outline fonts to arbitrary sizes and the availability of scalable outline fonts. Make sure that these new scalable fonts are available to your application by *not* setting the FPF_DESIGNED flag for **AvailFonts()** or **OpenDiskFont()**. Allow the user to create new font sizes by providing a way for her to manually enter the desired font size (the 1.3 OS returns the closest size, Release 2 returns the requested size).

See the Intuition and graphics library chapters, and the include file comments for additional information. See the ''Utility Library'' chapter for more information on **TagItems** and tag lists.

Conditional Code

Conditional code provides a second way to take advantage of Release 2 enhancements in a 1.3-compatible application. The basic idea is to add *low overhead* conditional code, based on library version, to make use of selected Release 2 features if they are available. There are some powerful and beneficial Release 2 features which are definitely worth conditional code.

The control flow for such conditional code is always based on whether a particular version of a library is available. Failure of **OpenLibrary()** (i.e., return value of NULL) means that the library version requested is not available. The version number of a library that successfully opened can be checked by testing **LibBase->lib_Version**. Always check for a version *greater or equal* to the version you need.

```
Examples of conditional library checking code:

/* Checking for presence of a new Release 2 library */
if( AslBase = OpenLibrary("asl.library", 37L) )
       { /* OK to use the ASL requester */ }
else
       { /* Must use a different method */ }

/* Checking version of an existing library with new Release 2 features */
if(GfxBase->lib_Version >= 37)   { /* then allow new genlock modes */}
```

ASL Requesters

The Release 2 ASL library provides standard file and font requesters. Allocation and use of an ASL requester can be handled by coding a simple subroutine to use the ASL requester if available. Otherwise use fallback code or a public domain requester. By now, many of you have probably coded your own requesters and you may be quite attached to them. In that case, at least give your users the option to use the ASL requester if they wish. By using the ASL requesters, you can provide a familiar interface to your users, gain the automatic benefit of all ASL file requester improvements, and stop maintaining your own requester code.

DOS System(), CreateNewProc(), and CON: Enhancements

If your program currently uses the 1.3 AmigaDOS **Execute()** or **CreateProc()** functions, then it is definitely worth conditional code to use their V37 replacements when running under Release 2. The **System()** function of Release 2 allows you to pass a command line to AmigaDOS as if it had been typed at a Shell window. **System()** can run synchronously with return values or asynchronously with automatic cleanup and it also sets up a proper *stdio* environment when passed a DOS filehandle for SYS_Input and NULL for SYS_Output. In combination with enhanced Release 2 CON: features, **System()** can provide a suitable execution environment on either Workbench or a custom screen. The **CreateNewProc()** function provides additional control and ease in process creation.

CON: input and output in custom Intuition screens and windows is now supported. New options in the Release 2 console handler (CON:) provide the ability to open a CON: on any public Intuition screen, or to attach a CON: to an existing Intuition window. Additional options can add a close gadget or create an AUTO console window which will only open if accessed for read or write. Add conditional code to use these system-supported methods when running under Release 2 or later versions of the OS. Note that additional CON: option keywords can be easily removed under 1.3 at runtime by terminating the CON: string with NULL after the window title. Consult *The AmigaDOS Manual* by Bantam Books for additional information on Release 2 CON: and DOS features.

The Display Database

The Release 2 graphics library and the Enhanced Chip Set (ECS) provide programmable display modes and enhanced genlock capabilities. Users with Release 2 and ECS may wish to use your application in one of the newer display modes. The Release 2 display database provides information on all of the display modes available with the user's machine and monitor. In addition, it provides useful information on the capabilities and aspect ratio of each mode (**DisplayInfo.Resolution.x** and to easily check if particular modes are available.

The **ExtNewScreen** structure used with Intuition's **OpenScreen()** function allows you to specify new display modes with the SA_DisplayID tag and a longword ModeID. The Release 2 graphics library **VideoControl()** function provides greatly enhanced genlock capabilities for machines with ECS and a genlock. Little conditional code is required to support these features. See the graphics library chapters and Autodocs for more information.

ARexx

Add conditional ARexx capabilities to your program. ARexx is available on all Release 2 machines, and many 1.3 users have purchased ARexx separately. ARexx capability adds value to your product and allows users and vertical market developers to create custom and hybrid applications. Add the ability to control your application externally via ARexx, and internally via ARexx macros. Allow the user to execute ARexx scripts to control other programs, including the ability to pass information from your program to other applications. For more information on adding ARexx functionality to your application, see the *Amiga Programmer's Guide to ARexx*, a publication by Commodore Applications and Technical Support (CATS). Contact your local Commodore support organization for information on ordering this book.

COMPATIBLE LIBRARIES

Compatible libraries provide a third method for using Release 2 while remaining 1.3-compatible. Some Release 2 libraries are 1.3-compatible and may be distributed with your product if you have a 1.3 Workbench License and an amendment to distribute the additional library.

IFFParse Library

The new IFFParse library is compatible with both Release 2 and the 1.3 version of the OS. IFFParse is a run-time library which provides low level code for writing, reading, and parsing IFF files. Use of IFFParse library and the new IFF example code modules can significantly reduce your development and debugging time. In addition, the IFFParse code modules provide effortless handing of the clipboard device. See the "IFFParse Library" chapter in this book and the IFF Appendix of the *Amiga ROM Kernel Reference Manual: Devices* for additional information.

Single Precision IEEE Math Libraries

The Release 2 single precision IEEE math libraries are also compatible with 1.3. These libraries provide single-precision math functions that will use a math coprocessor if available.

Third Party Compatible Libraries

Developers of new code may wish to take advantage of the ease with which a user interface can be created using the Release 2 GadTools and ASL support libraries. These new libraries are not 1.3-compatible but there are some third party development efforts towards providing 1.3-compatible versions of them. You may wish to explore this possibility.

Commodore Applications and Technical Support (CATS)

Commodore maintains a technical support group dedicated to helping developers achieve their goals with the Amiga. Currently, technical support programs are available to meet the needs of both smaller, independent software developers and larger corporations. Subscriptions to Commodore's technical support publication, Amiga Mail, is available to anyone with an interest in the latest news, Commodore software and hardware changes, and tips for developers.

To request an application for Commodore's developer support program, or a list of CATS technical publications send a self-addressed, stamped, 9" x 12" envelope to:

CATS-Information
1200 West Wilson Drive
West Chester, PA 19380-4231

Error Reports

In a complex technical manual, errors are often found after publication. When errors in this manual are found, they will be corrected in a subsequent printing. Updates will be published in Amiga Mail, Commodore's technical support publication.

Bug reports can be sent to Commodore electronically or by mail. Submitted reports must be clear, complete, and concise. Reports must include a telephone number and enough information so that the bug can be quickly verified from your report (i.e., please describe the bug and the steps that produced it).

Amiga Software Engineering Group
ATTN: BUG REPORTS
Commodore Business Machines
1200 Wilson Drive
West Chester, PA 19380-4231
USA

BIX: `amiga.com/bug.reports` (Commercial developers)
`amiga.cert/bug.reports` (Certified developers)
`amiga.dev/bugs` (Others)

USENET: `bugs@commodore.COM` or `uunet!cbmvax!bugs`

Chapter 2
INTUITION AND THE AMIGA GRAPHICAL USER INTERFACE

Intuition is the collective name for the function libraries, data structures and other elements needed to create a graphical interface for Amiga applications. Programmers use Intuition to perform user interface chores such as opening windows, managing menus, monitoring gadgets, reading the mouse position and so forth.

Newcomers to the Amiga sometimes think of Intuition as the Amiga's operating system but it is not. Intuition is just one component that together with Exec, AmigaDOS, and other subsystems make up the whole operating system. Intuition is the most *visible* part of the operating system though since it provides the graphical user interface familiar to all Amiga users.

About User Interfaces

What is a user interface? This sweeping phrase covers all aspects of communication between the user and the computer. It includes the innermost mechanisms of the computer and rises to the height of defining a philosophy to guide the interaction between human and machine. Intuition is, above all else, a philosophy turned into software.

Intuition's user interface philosophy is simple to describe: the interaction between the user and the computer should be consistent, simple and enjoyable; in a word, intuitive. Intuition supplies the tools needed to turn this philosophy into practice.

Implicit in this philosophy is the idea that the user interface should be graphical. A graphical user interface, or GUI, is a visually oriented method of communicating with a computer in which system resources are represented by pictorial symbols that can be manipulated with a pointing device such as a mouse. Other types of user interfaces are possible such as the Amiga's Shell in which text commands are entered by typing them at the keyboard. For more information about user interfaces, refer to the *Amiga User Interface Style Guide*.

ELEMENTS OF THE AMIGA GRAPHICAL USER INTERFACE SYSTEM

There is more to the Amiga user interface than Intuition. To build a complete user interface, application writers need to understand these other elements of the system software.

Table 2-1: Elements of the Amiga Graphical User Interface System

System Element	Purpose
Intuition	The main toolkit and function library for creating a graphical user interface (GUI) on the Amiga.
Workbench	The Amiga file system GUI in which icons represent applications and files.
Preferences	A family of editors and configuration files for setting Amiga system options.
BOOPSI	Subsystem of Intuition that allows applications to add extensions to Intuition through object-oriented techniques (Release 2 only).
Gadtools Library	A support library for creating Intuition gadgets and menus (Release 2 only).
ASL Library	A support library for creating Intuition requesters (Release 2 only).
Icon Library	Main library for using Workbench icons.
Workbench Library	A support library for Workbench icons and menus (Release 2 only).
Console Device	An I/O support module which allows windows to be treated as text-based virtual terminals.
Graphics Library	The main library of rendering and drawing routines.
Layers Library	A support library that manages overlapping, rectangular drawing areas which Intuition uses for windows.

As you read about Intuition in the chapters to follow, you will be introduced to each of these elements of the Amiga user interface in more detail.

GOALS OF INTUITION

Intuition was designed with two major goals in mind. The first is to give users a friendly and consistent environment to control the functions of the Amiga operating system and its applications.

The second goal (the big one) is to give application designers a graphical user interface toolkit that manages all the complexities of sharing the system with other programs that may be running at the same time. Since the Amiga is a multitasking computer, many programs can reside in memory at the same time sharing the system's resources with one another. Programs take turns running so that, from the user's point of view, it appears that many programs are running simultaneously.

On a multitasking computer like the Amiga, the user interface design must allow the user to control many programs with just one monitor, one keyboard, and one mouse. (Imagine driving many cars simultaneously with one steering wheel.) Intuition supplies the tools needed to solve this problem.

How the User Sees Intuition

Intuition solves the problem of interacting with multiple programs by dividing the display up into multiple screens and overlapping windows so that each application has its own work area. The user sees the Amiga environment through these windows, each of which can represent a different task or application context.

The user performs operations inside screens and windows with the mouse, a mechanical device that moves a pointer over the Amiga's display. The user moves the mouse to position the pointer on graphic symbols of various objects or actions. Buttons on the mouse are pressed to select or activate the item pointed to.

The user can switch back and forth between different jobs, such as writing a document, drawing an illustration, printing text, or getting help from the system simply by moving from one window to another with the mouse. With the mouse, the user can also change the shape and size of application windows, move them around on the screen, overlap them, bring a window to the foreground, and send a window to the background. By changing the arrangement of the windows, the user selects which information is visible and which application to work with next. (Screens may also be moved up or down in the display, and they can be moved in front of or behind other screens.)

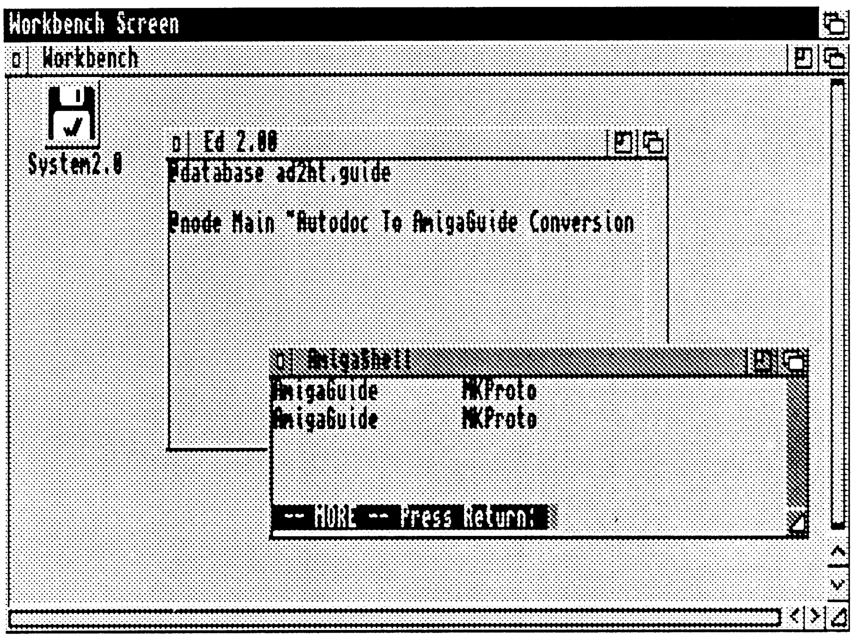


Figure 2-1: The Workbench Screen With Windows

WORKBENCH AND PREFERENCES

Normally, the Workbench screen (shown above) is the first screen the user sees upon booting the Amiga. Workbench is a special program supplied with every Amiga that gives the user a friendly and consistent graphic interface to the file system. It's the default environment the user starts out with.

In Workbench, disks, directories, files and other objects are symbolized by small pictures called icons which can be manipulated with the mouse. For instance, a program file can be executed by pointing to its icon with the mouse and double-clicking the left mouse button. The Workbench screen is automatically set up by Intuition and can be easily shared, so many application programs use it too.

User control of the OS is also supported through Preferences. Preferences is a family of editors and associated configuration files that allow the user to control the basic set up of the operating system. For example Printer Preferences sets up all the printer options.

Workbench, together with Preferences, gives the user an easy way to control the OS and launch applications. These programs are built with the same Intuition tools available to application programmers giving the whole Amiga system an integrated look and feel. Workbench and Preferences are important components of the Amiga graphic user interface system and are discussed in greater detail in later chapters.

INTUITION'S 3D LOOK

The Amiga operating system comes in different versions. The latest version, Release 2, contains significant improvements in the appearance of the Intuition graphical user interface, usually referred to as the *3D Look* of Intuition.

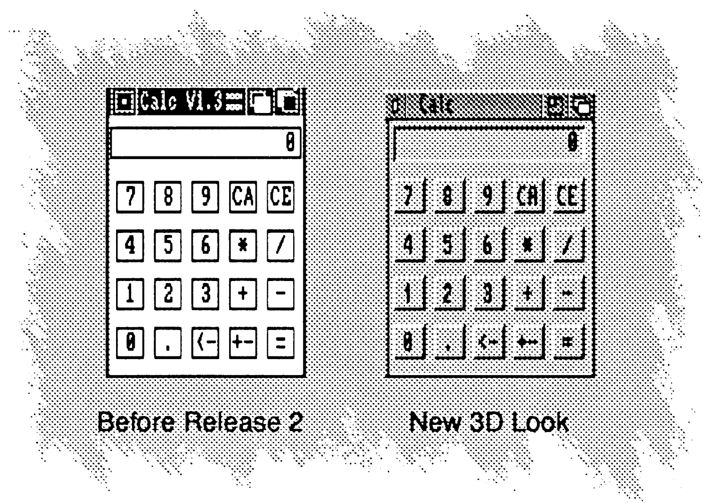


Figure 2-2: An Example of the 3D Look of Intuition

In the new 3D look of Intuition, objects are drawn so that light appears to come from the upper left of the display with shadows cast to the lower right. Using light and shadow gives the illusion of depth so that images appear to stand out or recede from the display. By convention, an image with a raised appearance indicates an object that is available for use or modifiable. An image with a recessed appearance indicates an object that is unmodifiable, or for display purposes only. Applications should follow the same conventions.

Release 2 has other improvements over 1.3 (V34) and earlier versions of the operating system. Among these are new display resolutions, display sizes, and new function libraries to support Intuition. Most of the examples listed in this book assume Release 2. Where appropriate, the old 1.3 methods are also described.

How an Application Sees Intuition

Intuition is organized as a library of over 100 functions. Before using an Intuition function you must first open the Intuition library. (In general, you must always open a library before you can call the functions of that library. See Chapter 1, "Introduction to Amiga System Libraries".)

COMPONENTS OF INTUITION

The types of data objects that the Intuition library functions create and control fall into six broad categories. These are the main components an application uses to build and operate a graphic user interface on the Amiga.

Table 2-2: GUI Components of Intuition

Screens	The display environment. Sets the resolution and number of colors.
Windows	A graphic rectangle within a screen representing a working context.
Menus	A list of choices displayed at the top of a screen that can be selected with the mouse.
Gadgets	A control symbolized by a graphic image that can be operated with the mouse or keyboard.
Requesters	Sub-windows for confirming actions, accessing files and other special options.
Input events	Mouse, keyboard or other input activity.

SCREENS AND WINDOWS

As mentioned earlier, Intuition allows multiple programs to share the display by managing a system of multiple *screens* and overlapping *windows*. A *screen* sets up the display environment and forms the background that application windows operate in. A *window* is simply a graphic rectangle that represents a work context. Each screen can have many windows on it.

Multiple screens and windows give each application its own separate visual context so that many programs can output graphics and text to the display at the same time without interfering with one another. Intuition (using the layers library) handles all the details of clipping graphics so they stay inside window bounds and remembering graphics that go temporarily out of sight when the user rearranges windows.

The keyboard and mouse are shared among applications through a simpler technique: *only one application window at a time can have the input focus.* Hence, Intuition ensures that only one window, called the *active* window gets to know about keyboard, mouse and other types of input activity.

Each application window is like a *virtual terminal* or *console.* Your program will seem to have the entire machine and display to itself. It can send text and graphics to its terminal window, and ask for input from any number of sources, ignoring the fact that other programs may be performing these same operations. Intuition handles all the housekeeping. In fact, your program can open several of these virtual terminals and treat each one as if it were the only program running on the machine. Intuition will keep track of all the activity and make sure commands and data are dispatched to the right place.

GADGETS, MENUS AND REQUESTERS

Intuition screens and windows provide an orderly way for multiple programs to share the display and input devices. Each application also needs a method for the user to send commands to it and select its options. Intuition supplies gadgets, menus and requesters for this purpose.

Gadgets

A *gadget* is an application control symbolized by a graphic image that can be operated with the mouse or keyboard. The imagery used for a gadget could look like a switch, a knob, a button, or just about anything. Intuition supplies some pre-fabricated gadgets, called *system gadgets*, for controlling window and screen arrangements. Other gadget types allow the user to select colors, enter text or numbers, and perform other simple operations.

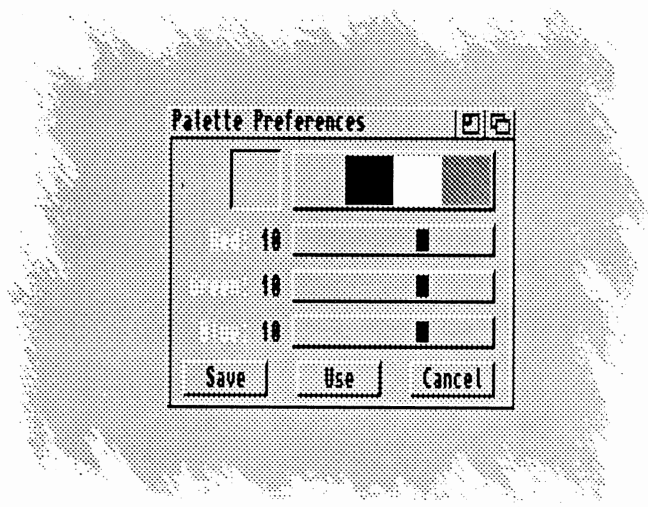


Figure 2-3: An Intuition Window with Gadgets

Most of the user's input for a typical Intuition application will be obtained with gadgets. Gadgets are discussed in detail in Chapter 5, ''Intuition Gadgets''. Additional information on programming gadgets for Release 2 of the operating system can be found in Chapter 15, ''GadTools Library''.

Menus

Intuition also supplies a menu system for accepting commands and options from the user. A *menu* is a list of choices displayed at the top of the screen from which the user can select with the mouse. Each screen has one *menu bar* that all application windows operating on the screen share. Whichever window is active controls what appears in the menu bar.

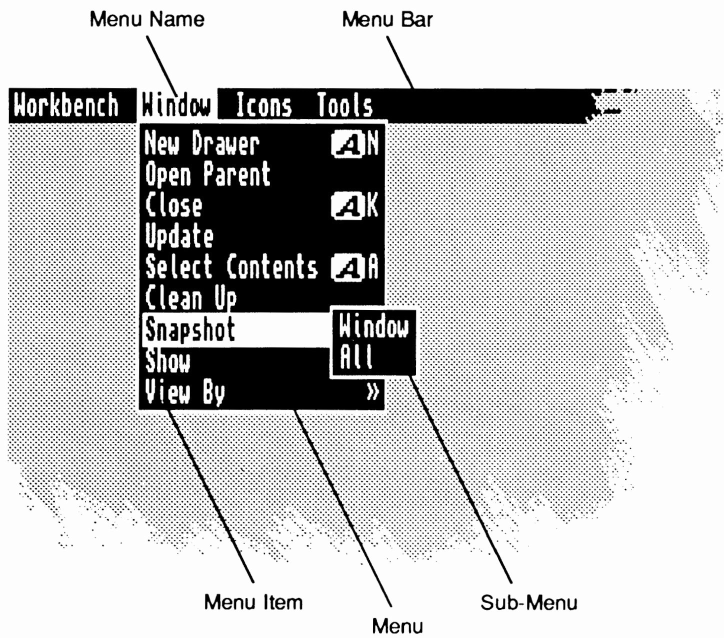


Figure 2-4: An Intuition Menu

The current set of menu choices can always be brought into view by pressing the right mouse button (the *menu button*) thus providing the user with a familiar landmark even in unfamiliar applications. Menus allow the user to browse through the possible set of actions that can be performed giving an outline-like overview of the functions offered by a program.

Menus are discussed in detail in Chapter 6, ''Intuition Menus''. Additional information on programming menus for Release 2 of the operating system can be found in Chapter 15, ''GadTools Library''.

Requesters

Gadgets and menus do much of the work of getting commands and option choices from the user. Sometimes though, an application needs to get further information from a user in response to a command which has already been initiated. In that case, a *requester* can be used. A *requester* is a temporary sub-window, usually containing several gadgets, used to confirm actions, access files, or adjust the special options of a command the user has already given.

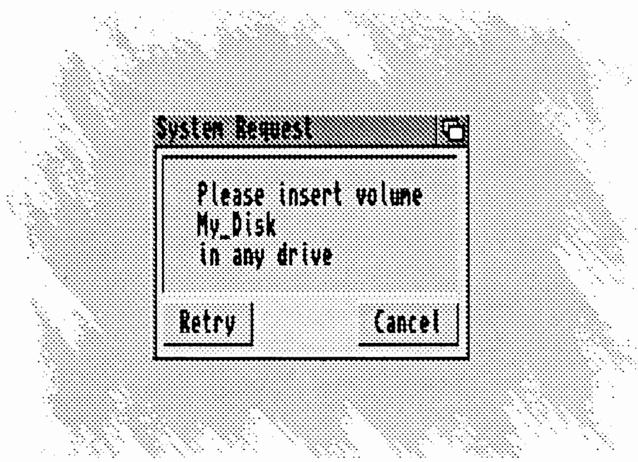


Figure 2-5: An Intuition Requester

Requesters are discussed in detail in Chapter 7, ''Intuition Requesters and Alerts''. Additional information on programming requesters for Release 2 of the system can be found in Chapter 16, ''ASL Library''.

The Intuition Input Event Loop

Once an application has set up the appropriate screen, window, gadgets menus and requesters, it waits for the user to do something. Intuition can notify an application whenever user activity occurs by sending a message. The message is simply a pointer to some memory owned by Intuition that contains an **IntuiMessage** data structure describing the user activity that occurred.

To wait for user activity or other events, the Exec library provides a special function named **Wait()**. The Exec **Wait()** function suspends your task allowing other applications or system tasks to run while your application is waiting for input or events from Intuition and other sources.

Thus, the basic outline for any Intuition program is:

- Set up the window, screen and any required gadgets, menus or requesters.
- **Wait()** for a message from Intuition about user activity or other events.
  Copy needed data from the message and tell Intuition you received it by replying.
  Look at the data and take the appropriate action.
- Repeat until the user wants to quit.

These steps, sometimes referred to as the *Intuition input event loop* are basically the same for any Intuition application.

As you might expect, Intuition can send a message to your application whenever the user presses a key on the keyboard or moves the mouse. Other types of input events Intuition will notify you about include gadget hits, menu item selection, time elapsing, disk insertion, disk removal, and window rearrangement.

Gadgets, menus, requesters are the nuts and bolts of the Intuition GUI toolkit. Much of the code in an application that uses Intuition deals with the set up and operation of these important data objects. No matter how simple, complex, or fanciful your program design, it will fit within the basic Intuition framework of windows and screens, gadgets, menus and requesters. The users of the Amiga understand these basic Intuition elements and trust that the building blocks remain constant. This consistency ensures that a well-designed program will be understandable to the naive user as well as to the sophisticate.

A Simple Intuition Program

The sample Intuition program that follows shows all of the basic requirements for an Intuition application. There are three important points:

- You must open the Intuition library before you can use the Intuition functions. Certain languages such as C require the pointer to the Intuition library to be assigned to a variable called **IntuitionBase** (see Chapter 1 for more about this).

- When you set up a window, you also specify the events that you want to know about. If the user performs some activity that triggers one of the events you specified, Intuition signals you and sends a message. The message is a pointer to an **IntuiMessage** data structure that describes the event in more detail. Messages about Intuition events are sent to a **MsgPort** structure which queues up the messages for you in a linked list so that you may respond to them at your convenience.

- Resources *must* be returned to the system. In this case, any windows, screens or libraries that were opened are closed before exiting.

EXAMPLE INTUITION EVENT LOOP

The Intuition event loop used in the example is very simple. The example first sets up a custom screen, opens a window on it, then waits for Intuition to send messages about user input with the following event loop:

```
winsignal = 1L << window1->UserPort->mp_SigBit;  /* window signal */
signalmask = winsignal;    /* example only waits for window events */

while( !done )  {
    signals = Wait(signalmask);
    if (signals & winsignal)
        done = handleIDCMP(window1);
}
```

Intuition sends messages about user activity to a special port known as the IDCMP. Each window can have its own IDCMP (in the code above the IDCMP is **window1->UserPort**). To wait for event messages to arrive at the IDCMP port, the example code calls the Exec **Wait()** function. It then processes and replies to any event messages that it gets in a subroutine named **handleIDCMP()**. For this example, the only event Intuition will report is the close window event. When the example detects this event, it closes the window, closes the screen, closes the Intuition library and exits. Event loops similar to this one are used in Intuition examples throughout this book. For more information about IDCMP and user input, see the chapters on ''Intuition Windows'' and ''Intuition Input and Output''.

INTUITION EXAMPLE (V36 AND LATER)

This example shows a simple Intuition program that works with Release 2 (V36) and later versions of the Amiga operating system.

```
/* easyintuition37.c -- Simple Intuition program for V37   */
/* (Release 2) and later versions of the operating system. */
/* Compiled with Lattice C v5.04: lc -L easyintuition37.c  */

#include <exec/types.h>             /* The Amiga data types file.         */
#include <intuition/intuition.h>    /* Intuition data strucutres, etc.    */
#include <graphics/displayinfo.h>   /* Release 2 Amiga display mode ID's  */
#include <libraries/dos.h>          /* Official return codes defined here */

#include <clib/exec_protos.h>       /* Exec function prototypes           */
#include <clib/intuition_protos.h>  /* Intuition function prototypes      */

/* Force use of new variable names to help prevent errors  */
#define INTUI_V36_NAMES_ONLY

#ifdef LATTICE                      /* Disable Ctrl-C handling in SAS/C   */
int CXBRK(void)   {return(0);}
void chkabort(void) {return;}
#endif

/* Use lowest non-obsolete version that supplies the functions needed. */
#define INTUITION_REV 37

/* Declare the prototypes of our own functions.  Prototypes for system */
/* functions are declared in the header files in the clib directory.   */
VOID cleanExit( struct Screen *, struct Window *, LONG );
BOOL handleIDCMP( struct Window *);

struct Library *IntuitionBase = NULL;

/* Position and sizes for our window */
#define WIN_LEFTEDGE   20
#define WIN_TOPEDGE    20
#define WIN_WIDTH     400
#define WIN_MINWIDTH   80
#define WIN_HEIGHT    150
#define WIN_MINHEIGHT  20


VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, winsignal, signals;
    BOOL done = FALSE;
    UWORD pens[]={~0};

    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open the Intuition Library */
    IntuitionBase = OpenLibrary( "intuition.library",INTUITION_REV );
    if (IntuitionBase == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Open any other required libraries and make */
    /* any assignments that were postponed above  */

    /* Open the screen */
    screen1 = OpenScreenTags(NULL,
                         SA_Pens,   (ULONG)pens,
                         SA_DisplayID, HIRES_KEY,
                         SA_Depth, 2,
                         SA_Title, (ULONG)"Our Screen",
                         TAG_DONE);

    if (screen1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* ... and open the window */
```

```
    window1 = OpenWindowTags(NULL,
                        /* Specify window dimensions and limits */
                        WA_Left,         WIN_LEFTEDGE,
                        WA_Top,          WIN_TOPEDGE,
                        WA_Width,        WIN_WIDTH,
                        WA_Height,       WIN_HEIGHT,
                        WA_MinWidth,     WIN_MINWIDTH,
                        WA_MinHeight,    WIN_MINHEIGHT,
                        WA_MaxWidth,     ~0,
                        WA_MaxHeight,    ~0,
                        /* Specify the system gadgets we want */
                        WA_CloseGadget,  TRUE,
                        WA_SizeGadget,   TRUE,
                        WA_DepthGadget,  TRUE,
                        WA_DragBar,      TRUE,
                        /* Specify other attributes           */
                        WA_Activate,     TRUE,
                        WA_NoCareRefresh,TRUE,

                        /* Specify the events we want to know about */
                        WA_IDCMP,        IDCMP_CLOSEWINDOW,

                        /* Attach the window to the open screen ...*/
                        WA_CustomScreen, screen1,
                        WA_Title,        "EasyWindow",
                        WA_ScreenTitle,  "Our Screen - EasyWindow is Active",
                        TAG_DONE);
    if (window1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Set up the signals for the events we want to hear about ...   */
    winsignal = 1L << window1->UserPort->mp_SigBit;  /* window IDCMP */
    signalmask = winsignal;   /* we are only waiting on IDCMP events */

    /* Here's the main input event loop where we wait for events.    */
    /* We have asked Intuition to send us CLOSEWINDOW IDCMP events   */
    /* Exec will wake us if any event we are waiting for occurs.     */
    while( !done )
    {
        signals = Wait(signalmask);

        /* An event occurred - now act on the signal(s) we received.*/
        /* We were only waiting on one signal (winsignal) in our    */
        /* signalmask, so we actually know we received winsignal.   */
        if(signals & winsignal)
            done = handleIDCMP(window1);    /* done if close gadget */
    }
    cleanExit(screen1, window1, RETURN_OK); /* Exit the program     */
}

BOOL handleIDCMP( struct Window *win )
{
    BOOL done = FALSE;
    struct IntuiMessage *message = NULL;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
    {
        class = message->Class;  /* get all data we need from message */

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);

        /* See what events occurred */
        switch( class )
        {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            default:
                break;
        }
    }
    return(done);
}
```

```
VOID cleanExit( struct Screen *scrn, struct Window *wind, LONG returnValue )
{
    /* Close things in the reverse order of opening */
    if (wind) CloseWindow( wind );      /* Close window if opened */
    if (scrn) CloseScreen( scrn );      /* Close screen if opened */

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( IntuitionBase );
    exit(returnValue);
}
```

INTUITION EXAMPLE (ALL VERSIONS)

Here's the same example as above written for both Release 2 and earlier versions of the operating system. The main difference here is that this example avoids using any new Release 2 functions, but does pass extended structures to the older Intuition functions so that some new Release 2 features may be accessed in a backward-compatible manner.

```
/* easyintuition.c  Simple backward-compatible V37 Intuition example    */
/*                                                                      */
/* This example uses extended structures with the pre-V37 OpenScreen()  */
/* and OpenWindow() functions to compatibly open an Intuition display.  */
/* Enhanced V37 options specified via tags are ignored on 1.3 systems.  */
/* Compiled with Lattice C v5.10: lc -L easyintuition.c                 */

/* Force use of new variable names to help prevent errors  */
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>             /* The Amiga data types file.         */
#include <intuition/intuition.h>    /* Intuition data strucutres, etc.    */
#include <libraries/dos.h>          /* Official return codes defined here */

#include <clib/exec_protos.h>       /* Exec function prototypes           */
#include <clib/intuition_protos.h>  /* Intuition function prototypes      */

#ifdef LATTICE                      /* Disable Ctrl-C handling in SAS/C   */
int CXBRK(void)   {return(0);}
void chkabort(void) {return;}
#endif

/* Use lowest non-obsolete version that supplies the functions needed. */
#define INTUITION_REV 33L

/* Declare the prototypes of our own functions. Prototypes for system  */
/* functions are declared in the header files in the clib directory    */
VOID cleanExit( struct Screen *, struct Window *, LONG );
BOOL handleIDCMP( struct Window *);

struct Library *IntuitionBase = NULL;

/* We can specify that we want the V37-compatible 3D look when
 * running under V37 by adding an SA_Pens tag.
 */
WORD pens[] = {~0};     /* empty pen array to get default 3D look */
struct TagItem ourscreentags[] = {
      { SA_Pens, (ULONG)pens },
      { TAG_DONE }};

/* ExtNewScreen is an extended NewScreen structure.
 * NS_EXTENDED flags that there is a tag pointer to additional
 * tag information at the end of this structure.  The tags will
 * be parsed by Release 2 but ignored by earlier OS versions.
 */
struct ExtNewScreen fullHires =
    {
    0,                  /* LeftEdge must be zero prior to Release 2 */
    0,                  /* TopEdge */
    640,                /* Width (high-resolution) */
    STDSCREENHEIGHT,    /* Height (non-interlace)  */
    2,                  /* Depth (4 colors will be available) */
    0,1,                /* Default DetailPen and BlockPen  */
```

```
    HIRES,              /* the high-resolution display mode */
    CUSTOMSCREEN | NS_EXTENDED,      /* the screen type */
    NULL,               /* no special font */
    "Our Screen",       /* the screen title */
    NULL,               /* no custom screen gadgets (not supported) */
    NULL,               /* no CustomBitMap */
    ourscreentags       /* tags for additional V37 features */
    };

/* Position and sizes for our window */
#define WIN_LEFTEDGE   20
#define WIN_TOPEDGE    20
#define WIN_WIDTH     400
#define WIN_MINWIDTH   80
#define WIN_HEIGHT    150
#define WIN_MINHEIGHT  20

/* Under V37, we'll get a special screen title when our window is active */
UBYTE activetitle[] = {"Our Screen - EasyWindow is Active"};

struct TagItem ourwindowtags[] = {
      { WA_ScreenTitle, (ULONG)&activetitle[0] },
      { TAG_DONE }};

/* ExtNewWindow is an extended NewWindow structure.
 * NW_EXTENDED indicates that there is a tag pointer to additional tag
 * information at the end of this structure.  The tags will be parsed
 * by Release 2 but ignored by earlier OS versions.
 */
struct ExtNewWindow easyWindow =
    {
    WIN_LEFTEDGE,
    WIN_TOPEDGE,
    WIN_WIDTH,
    WIN_HEIGHT,
    -1,-1,              /* Means use the screen's Detail and Block pens    */

    IDCMP_CLOSEWINDOW, /* This field specifies the events we want to get */

    /* These flags specify system gadgets and other window attributes    */
    /* including the EXTENDED flag which flags this as an ExtNewWindow   */
    WFLG_CLOSEGADGET | WFLG_SMART_REFRESH | WFLG_ACTIVATE | WFLG_DRAGBAR |
    WFLG_DEPTHGADGET | WFLG_SIZEGADGET   | WFLG_NOCAREREFRESH |
    WFLG_NW_EXTENDED,

    NULL,               /* Pointer to the first gadget  */
    NULL,               /* No checkmark.                */
    "EasyWindow",       /* Window title.                */
    NULL,               /* Attach a screen later.       */
    NULL,               /* Let Intuition set up BitMap  */
    WIN_MINWIDTH,       /* Minimum width.       */
    WIN_MINHEIGHT,      /* Minimum height.      */
    -1,                 /* Maximum width (screen size)  */
    -1,                 /* Maximum height (screen size) */
    CUSTOMSCREEN,       /* A screen of our own. */
    ourwindowtags       /* tags for additional V37 features */
    };


VOID main(int argc, char *argv[])
{
    /* Declare variables here */
    ULONG signalmask, winsignal, signals;
    BOOL done = FALSE;
    struct Screen *screen1 = NULL;
    struct Window *window1 = NULL;

    /* Open Intuition Library.  NOTE - We are accepting version 33 (1.2)
     * or higher because we are opening our display in a compatible manner.
     * However - If you add to this example, do NOT use any NEW V37
     * functions unless IntuitionBase->lib_Version is >= 37
     */
    IntuitionBase = OpenLibrary( "intuition.library",INTUITION_REV );
    if (IntuitionBase == NULL)
        cleanExit(screen1, window1, RETURN_WARN);
```

```
    /* Open any other required libraries and make */
    /* any assignments that were postponed above  */

    /* Open the screen */
    screen1 = OpenScreen(&fullHires);
    if (screen1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Attach the window to the open screen ... */
    easyWindow.Screen = screen1;

    /* ... and open the window */
    window1 = OpenWindow(&easyWindow);
    if (window1 == NULL)
        cleanExit(screen1, window1, RETURN_WARN);

    /* Set up the signals for the events we want to hear about ...   */
    winsignal = 1L << window1->UserPort->mp_SigBit;  /* window IDCMP */
    signalmask = winsignal;     /* we will only wait on IDCMP events */

    /* Here's the main input event loop where we wait for events.    */
    /* We have asked Intuition to send us CLOSEWINDOW IDCMP events   */
    /* Exec will wake us if any event we are waiting for occurs.     */
    while( !done )
    {
        signals = Wait(signalmask);

        /* An event occurred - now act on the signal(s) we received.*/
        /* We were only waiting on one signal (winsignal) in our    */
        /* signalmask, so we actually know we received winsignal.   */
        if(signals & winsignal)
            done = handleIDCMP(window1);    /* done if close gadget */
    }
    cleanExit(screen1, window1, RETURN_OK); /* Exit the program     */
}


BOOL handleIDCMP( struct Window *win )
{
    BOOL done = FALSE;
    struct IntuiMessage *message;
    ULONG class;

    /* Examine pending messages */
    while( message = (struct IntuiMessage *)GetMsg(win->UserPort) )
    {
        class = message->Class;   /* get all data we need from message */

        /* When we're through with a message, reply */
        ReplyMsg( (struct Message *)message);

        /* See what events occurred */
        switch( class )
        {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            default:
                break;
        }
    }
    return(done);
}


VOID cleanExit( struct Screen *scrn, struct Window *wind, LONG returnValue )
{
    /* Close things in the reverse order of opening */
    if (wind) CloseWindow( wind );      /* Close window if opened */
    if (scrn) CloseScreen( scrn );      /* Close screen if opened */

    /* Close the library, and then exit */
    if (IntuitionBase) CloseLibrary( IntuitionBase );
    exit(returnValue);
}
```

Chapter 3
INTUITION SCREENS

Intuition screens are the basis of any display Intuition can make. Screens determine the fundamental characteristics of the display such as the resolution and palette and they set up the environment for multiple, overlapping windows that makes it possible for each application to have its own separate visual context. This chapter shows how to use existing screens and how to create new screens.

Types of Screens

Screens are important because they determine the basic resolution and maximum number of colors in the display. Once a screen is set up, these attributes cannot be changed so any graphics work done on a given screen is restricted to that screen's resolution and number of colors. Hence, the type of screen used is a basic design decision.

With Intuition screens, a video display can be created in any one of the many Amiga *display modes*. The basic parameters of the video display such as resolution, total size, frame rate, genlock compatibility, support of screen movement and number of colors are defined by these modes. There are currently four basic modes available on all Amiga models. These basic modes work with conventional monitors (15 kHz scan rate) and older versions of the operating system.

Basic Amiga	**Resolution**		**Maximum**	**Supports**
Display Modes	**NTSC**	**PAL**	**Colors**	**HAM/EHB***
Lores	320x200	320x256	32 of 4096	Yes
Lores-Interlaced	320x400	320x512	32 of 4096	Yes
Hires	640x200	640x256	16 of 4096	No
Hires-Interlaced	640x400	640x512	16 of 4096	No

*HAM and EHB modes provide for additional colors with some restrictions.

Table 3-1: Basic Amiga Display Modes

With Release 2 of the operating system, many other display modes are available (these usually require a special monitor or ECS). All these display modes, including the specialized modes, are integrated through the graphics library *display database*. See the "Graphics Primitives" chapter for a complete list of all Amiga display modes.

MULTIPLE SCREENS

All Intuition display objects (such as windows and menus) take graphical characteristics from the screen. These objects are restricted to the same resolution and maximum number of colors as the screen they operate in. Other characteristics such as the palette, pens and fonts are inherited from the screen (but may be changed on a case by case basis).

This is not too much of a restriction because the Amiga can maintain multiple screens in memory at the same time. In other words, one application can use a high resolution screen (with 16 colors) while another application uses a low resolution screen (with 32 colors) at the same time. Screens typically take up the entire viewing area so only one is usually visible. But screens can be moved up and down or rearranged allowing the user (or application) to move between screens easily.

PUBLIC SCREENS AND CUSTOM SCREENS

An application may choose to use an existing screen or to create its own screen. For instance, the normal Amiga startup process opens the Workbench screen (Workbench is the Amiga's default user interface). Any application is free to use the Workbench screen instead of opening a new one. Screens that can be shared this way are called *public* screens.

Public screens are a new feature of Release 2 (V36). In older versions of the OS, only the Workbench screen could be shared. Now any screen may be set up as a public screen so that other applications may use it.

The use of an existing public screen, like the Workbench screen, requires little effort by the application and does not use up any memory. However, using Workbench or another existing public screen means some flexibility is lost; the resolution, maximum number of colors and other attributes are already set. If the application cannot function under these limitations, it may open its own *custom* screen.

Custom screens allow for complete control of the display space so an application can get exactly the kind of display it wants. However, since creating a new, custom screen uses up memory, they should only be used when there are no suitable public screens available.

Owners of a custom screen can keep their screen private, or they may allow other applications to share their screen by registering the screen with the operating system as a public screen. See the section on ''Public Screen Functions'' later in this chapter for more about public screens and Workbench.

SCREEN COMPONENTS

Screens have very little visual impact, they simply provide a resolution specific area to place other objects such as windows and menus. Screens have no borders. Only the title bar marks the screen limits (specifying the left and top edges, and the width of the screen), and the title bar may be hidden, or obscured by graphics or windows.

The title bar also serves as the menu bar when the user presses the menu button on the mouse. The menu bar area is shared by all applications operating within the screen.

Within the title bar, there are two gadgets: a screen drag gadget and a depth-arrangement gadget. The screen drag gadget allows the screen to be moved up and down. The depth-arrangement gadget allows the screen to be placed in front or behind all other screens.

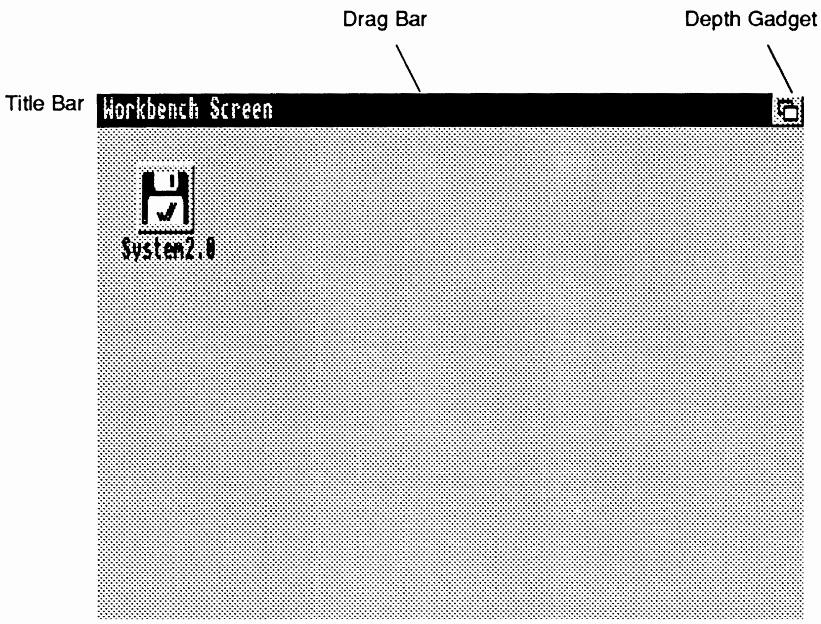


Figure 3-1: An Intuition Screen (Workbench)

Screens are always rectangular, and the areas at the sides and bottom of the display that are not within the screen's limits are filled with the background color. The area above all visible screens is filled with the background color of the highest screen. These areas surrounding the screen (normally unused) are known as the *overscan* area. The Amiga display system allows the overscan area to be used for graphics under special circumstances (see the section on ''Overscan and the Display Clip'' later in this chapter).

Screen Data Structures

The Amiga uses color registers and bitplane organization as its internal representation of display data. Screens require a color table and display raster memory for each bitplane. This is the memory where imagery is rendered and later translated by the hardware into the actual video display. This information is contained in data structures from the Amiga's graphics library.

A **ViewPort** is the main data structure used by the graphics library to represent a screen. Pointers to each of the screen's bitplanes are stored in the graphics library **BitMap** structure. Color table information is stored in a graphics structure called a **ColorMap**. And the screen's drawing and font information is stored in the **RastPort** structure.

The graphics **RastPort** structure is a general-purpose handle that the graphics library uses for drawing operations. Many Intuition drawing functions also take a **RastPort** address as a parameter. This makes sense since the **RastPort** structure contains drawing variables as well as a pointer to the **BitMap** telling where to draw. See the ''Graphics Primitives'' chapter for more information on these structures and how they are used.

THE INTUITION SCREEN DATA STRUCTURE

The structures mentioned above are unified along with other information in Intuition's **Screen** data structure defined in the include file *<intuition/screens.h>*. Notice that the **Screen** structure contains instances of a **ViewPort**, **RastPort** and **BitMap**.

```
struct Screen
    {
    struct Screen *NextScreen;
    struct Window *FirstWindow;
    WORD LeftEdge, TopEdge, Width, Height;
    WORD MouseY, MouseX;
    UWORD Flags;
    UBYTE *Title, *DefaultTitle;
    BYTE BarHeight, BarVBorder, BarHBorder, MenuVBorder, MenuHBorder;
    BYTE WBorTop, WBorLeft, WBorRight, WBorBottom;
    struct TextAttr *Font;
    struct ViewPort ViewPort;
    struct RastPort RastPort;
    struct BitMap BitMap;
    struct Layer_Info LayerInfo;
    struct Gadget *FirstGadget;
    UBYTE DetailPen, BlockPen;
    UWORD SaveColor0;
    struct Layer *BarLayer;
    UBYTE *ExtData, *UserData;
    };
```

In general, applications don't need to access the fields in the **Screen** structure directly; they use Intuition functions to manipulate the screen instead. Likewise, applications do not set up the **Screen** themselves; they use one of the **OpenScreen()** calls (see below). Here is a description of some of the more interesting members of the **Screen** structure (it is not meant to be a complete description of all the fields).

LeftEdge, TopEdge

The **LeftEdge** and **TopEdge** variables give the position of the screen relative to the upper left corner of the monitor's visible display (as set by the user in the Overscan preferences editor). If it is positioned down or to the right, the values are positive. If the screen is positioned up or to the left, the values are negative. The values are in screen resolution pixels. In systems prior to V36, **LeftEdge** positioning is ignored and negative **TopEdge** positions are illegal.

The screen position may be set when the screen is opened or later by calling the **MoveScreen()** function. Note that the screen's actual display position may not exactly equal the coordinates given in the **LeftEdge** and **TopEdge** fields of the **Screen** structure. This can cause a window which is opened in the visible part of the screen to be incorrectly positioned by a few pixels in each direction. This complication is due to hardware constraints that limit the fineness of screen positioning. For instance, high resolution screens can only be positioned in low resolution pixel coordinates, yet the values in the **LeftEdge** and **TopEdge** use high resolution pixel coordinates. So when the screen is displayed, its position is rounded to a position available for the monitor.

MouseX, MouseY

Position of the mouse with respect to the upper left corner of the screen.

ViewPort, RastPort, BitMap, LayerInfo

Actual instances of the graphics library data structures associated with this screen (*not* pointers to structures). For normal use of custom screens, these structures may be ignored.

BarLayer

A pointer to the **Layer** structure for the screen's title bar.

WBorTop, WBorLeft, WBorRight, WBorBottom
Window border values, see the "Intuition Windows" chapter for information on pre-calculating the size of window borders for windows that open in this screen.

Font
The default screen font, this can be used to pre-calculate the size of the window borders for windows that open in this screen.

UserData
Free for application use.

Other **Screen** structure members provide information on the title bar layer, and attributes of menus and windows opened in the screen. Of particular interest are the values that allow precalculation of window border size. These variables will be discussed in the chapter "Intuition Windows".

OTHER SCREEN DATA STRUCTURES

In addition to the **Screen** structure, Intuition uses some other supporting structures defined in the include file *<intuition/screens.h>* and in *<utility/tagitems.h>*. (See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a complete listing.)

Table 3-2: Data Structures Used with Intuition Screens

Structure Name	Description	Defined in Include File
Screen	Main Intuition structure that defines a screen (see above)	*<intuition/screens.h>*
DrawInfo	Holds the screen's pen, font and aspect data for Intuition	*<intuition/screens.h>*
TagItem	General purpose parameter structure used to set up screens in V36	*<utility/tagitem.h>*
NewScreen	Parameter structure used to create a screen in V34	*<intuition/screens.h>*
ExtNewScreen	An extension to the **NewScreen** structure used in V37 for backward compatibility with older systems	*<intuition/screens.h>*

As previously mentioned, there is an Intuition **Screen** structure (and a corresponding graphics **ViewPort**) for every screen in memory. Under Release 2, whenever a new screen is created, Intuition also creates an auxiliary data structure called a **DrawInfo**.

The **DrawInfo** structure is similar to a **RastPort** in that it holds drawing information. But where a **RastPort** is used at the lower graphics level, the **DrawInfo** structure is used at the higher Intuition level. Specifically, **DrawInfo** contains data needed to support the *New Look* of Intuition in Release 2. (See the section below, "DrawInfo and the 3D Look", for more information.)

Another new feature of Release 2 is *tag items*. A **TagItem** is a general purpose parameter structure used to pass arguments to many of the functions in the Release 2 system software. Each tag consists of a LONG tag ID (**ti_Tag**) and a LONG tag data value (**ti_Data**). With screens, tag items are used to describe the attributes an application wants for a new, custom screen. Tag items replace the **NewScreen** structure, the set of parameters used in older versions of the OS to set up a screen.

Applications may wish to use tag items to set up a new screen instead of the **NewScreen** structure since tag items are often more convenient. For the sake of backwards compatibility, the **ExtNewScreen** structure is available. **ExtNewScreen** combines the **NewScreen** method used to define screens in older versions of the OS with the tag item method used in Release 2. The examples listed in the next section show how these various data structures can be used to set up a new screen.

Custom Screen Functions

All applications require a screen to work in. This can be an existing, public screen or a new, custom screen created by the application itself. To create a new, custom screen to work with, you call **OpenScreen()** or one of its variants.

Table 3-3: Custom Screen Functions

OpenScreenTags() **OpenScreenTagList()**	Create a new, custom screen from a tag list. Use either one of these with Release 2 (V36) or later versions of the OS.
OpenScreen()	Create a new, custom screen from an **ExtNewScreen** structure. Use this if your application must be compatible with 1.3 (V34) or earlier versions of the operating system.
CloseScreen()	Close a custom screen and free the memory it used.

CREATING A NEW CUSTOM SCREEN

There are three functions you can use to open a custom screen: **OpenScreen()**, **OpenScreenTags()** or **OpenScreenTagList()**. Prior to Release 2 (V36), **OpenScreen()** was used to create a new screen. With V36 and later versions of the operating system, this call is superseded by **OpenScreenTagList()** and **OpenScreenTags()**.

```
struct Screen *OpenScreen( struct NewScreen *)
struct Screen *OpenScreenTagList( struct NewScreen * , struct TagItem *)
struct Screen *OpenScreenTags( struct NewScreen *, ULONG, ULONG, ... )
```

The old **OpenScreen()** call relied on a fixed size data structure (**NewScreen**) which made little allowance for extensions and growth. The new calls are tag based, allowing for the addition of new features without modification of existing structures and applications. The ''Screen Attributes'' section below contains a complete list of all the tag options available for setting up an Intuition screen. For a general description of tag items, see the ''Utility Library'' chapter.

A Custom Screen Example

There are so many tag options available with screens it can be a bit overwhelming. Before discussing more details, it may be helpful to look at a simple example. The code below opens a new, custom screen using the **OpenScreenTags()** call. The example uses just two tag items (SA_Depth and SA_Pens) which provide the minimum attributes needed to make a screen that uses the new 3D look of Intuition available in Release 2. (See the section on ''DrawInfo and the 3D Look'' below for more information.)

```
/* newlookscreen.c
** open a screen with the "new look".
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y newlookscreen
** blink LIB:c.o newlookscreen.o TO newlookscreen LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library *IntuitionBase;

/* Simple routine to demonstrate opening a screen with the new look.
** Simply supply the tag SA_Pens along with a minimal pen specification,
** Intuition will fill in all unspecified values with defaults.
** Since we are not supplying values, all are Intuition defaults.
*/
VOID main(int argc, char **argv)
{
UWORD pens[] = { ~0 };
struct Screen *my_screen;

IntuitionBase = OpenLibrary("intuition.library",0);
if (NULL != IntuitionBase)
    {
    if (IntuitionBase->lib_Version >= 37)
        {
        /* The screen is opened two bitplanes deep so that the
        ** new look will show-up better.
        */
        if (NULL != (my_screen = OpenScreenTags(NULL,
                                SA_Pens, (ULONG)pens,
                                SA_Depth, 2,
                                TAG_DONE)))
            {
            /* screen successfully opened */
            Delay(30L);  /* normally the program would be here */

            CloseScreen(my_screen);
            }
        }
    CloseLibrary(IntuitionBase);
    }
}
```

The example above runs only under Release 2 (V36) and later versions of the OS. To make a custom screen that works under both Release 2 and earlier versions of the operating system, use the original **OpenScreen()** function.

The **NewScreen** structure used with **OpenScreen()** has been extended with a tag list in V36 to form an **ExtNewScreen**. This is done by setting the NS_EXTENDED bit in the **Type** field of the **NewScreen** structure and adding a pointer to an array of tags to the end of the structure. The NS_EXTENDED bit is ignored in older versions of the operating system, so the tags can be transparently added to existing applications and the features will appear when executed in a system running V36 or greater. See the **OpenScreen()** Autodocs and the include file *<intuition/screens.h>* for more information on NS_EXTENDED and the **ExtNewScreen** structure.

Creating A Custom Screen that Works With Older Systems

Here's an example of how to use the old **OpenScreen()** call with an **ExtNewScreen** structure to make a new, custom screen under any version of the Amiga operating system. If the version is V36 or later, additional Release 2 features specified via tags, in this case the new 3D look of Intuition, will be incorporated in the window.

```
;/* screen34to37.c - Execute me to compile me with SAS 5.10
LC -b1 -cfistq -v -y -j73 screen34to37.c
blink FROM LIB:c.o screen34to37.o TO screen34to37 LIB LIB:lc.lib LIB:amiga.lib
quit
*/

#define INTUI_V36_NAMES_ONLY         /* We'll use the newer Intuition names. */

#include <exec/types.h>              /* Amiga data types.              */
#include <intuition/intuition.h>     /* Lots of important Intuition    */
#include <intuition/screens.h>       /* structures we will be using.   */

#include <clib/exec_protos.h>        /* Function prototypes            */
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }   /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }   /* really */
#endif

struct Library *IntuitionBase;       /* Intuition library base          */

/* Simple example to show how to open a custom screen that gives the new look
 * under V37, yet still works with older version of the operating system.
 * Attach the tag SA_Pens and a minimal pen specification to ExtNewScreen,
 * and call the old OpenScreen() function.  The tags will be ignored by
 * V34 and earlier versions of the OS.  In V36 and later the tags are
 * accepted by Intuition.
*/

VOID main(int argc, char **argv)
{
UWORD pens[] = { ~0 };               /* This is the minimal pen specification*/
struct Screen      *my_screen;       /* Pointer to our new, custom screen    */
struct ExtNewScreen myscreen_setup;  /* Same as NewScreen with tags attached */
struct TagItem      myscreen_tags[2];/* A small tag array                    */

/* Open the library before you call any functions */
IntuitionBase = OpenLibrary("intuition.library",0);
if (NULL != IntuitionBase)
   {
      /* Fill in the tag list with the minimal pen specification */
      myscreen_tags[0].ti_Tag=SA_Pens;
      myscreen_tags[0].ti_Data=(ULONG) pens;
      myscreen_tags[1].ti_Tag=TAG_DONE;

      /* The screen is opened two bitplanes deep so that the
      ** new look will show-up better.
      **/
      myscreen_setup.LeftEdge=0;
      myscreen_setup.TopEdge=0;
      myscreen_setup.Width=640;                 /* Smaller values here reduce   */
      myscreen_setup.Height=STDSCREENHEIGHT;    /* drawing area and save memory.*/
      myscreen_setup.Depth=2;                   /* Two planes means 4 colors.   */
      myscreen_setup.DetailPen=0;               /* Normal V34 pen colors.       */
      myscreen_setup.BlockPen=1;
      myscreen_setup.ViewModes=HIRES;
      myscreen_setup.Type=CUSTOMSCREEN | NS_EXTENDED; /* Extended NewScreen flag */
      myscreen_setup.Font=NULL;
      myscreen_setup.DefaultTitle="My Screen";
      myscreen_setup.Gadgets=NULL;
      myscreen_setup.CustomBitMap=NULL;
      /* Attach the pen specification tags to the ExtNewScreen structure */
      myscreen_setup.Extension=myscreen_tags;
```

```
    if (NULL != (my_screen =
      OpenScreen((struct NewScreen *)&myscreen_setup)));
        {
        /* screen successfully opened */

        Delay(200L);  /* normally the program would be here */

        CloseScreen(my_screen);
        }
    CloseLibrary(IntuitionBase);
  }
}
```

As you can see from the examples above, there are many ways to create a new, custom screen. Further references to "**OpenScreenTagList()**" in this manual are referring to any one of the three calls: **OpenScreenTagList()**, **OpenScreenTags()**, or **OpenScreen()** used with tags in an **ExtNewScreen** as shown above.

Return Values from OpenScreenTagList()

OpenScreenTagList() and its variants return a pointer to a **Screen** structure on the successful creation of a new screen and NULL on failure. With V36, the call now supports extended error codes on failure. The error returns provide information on the type of failure, giving the application a greater chance of recovery. To get the extended error code, you need to use the SA_ErrorCode tag; the code itself will be placed into the LONG pointed to by the **TagItem.ti_Data field. Here are the codes:**

OSERR_NOMONITOR
: The monitor needed to display the requested mode is not available. An example of this error would be opening a Productivity mode screen on a system without a VGA or multisync monitor.

OSERR_NOCHIPS
: Newer custom chips are required for this screen mode. For instance, the ECS Denise is required for the productivity modes.

OSERR_NOMEM
: Could not allocate enough memory.

OSERR_NOCHIPMEM
: Could not allocate enough Chip memory.

OSERR_PUBNOTUNIQUE
: Could not create public screen--name already used. The system requires that public screen names be unique.

OSERR_UNKNOWNMODE
: Display mode requested was not recognized. The system does not understand the value specified with the SA_DisplayID tag.

Closing the Screen

When an application has finished using a screen, the memory that the screen occupied should be returned to the system by calling **CloseScreen()**. Normally, an application should close *only* those screens that it created. Under V34 and earlier versions of the OS, **CloseScreen()** returns no values. Under Release 2, **CloseScreen()** returns a boolean value, TRUE for success and FALSE for failure. **CloseScreen()** can fail if the screen is public and another task is still using the screen.

SCREEN ATTRIBUTES

The sections above discuss only the basic functions and screen types that Intuition programmers need to understand to create a custom screen. Intuition supports an astonishing number of additional display features and options. In this section and the sections to follow, the finer points of screen attributes and the functions that control them are presented.

Screen attributes are specified using the tag item scheme described in the "Utility Library" chapter. Therefore, the screen attributes are listed here by tag values. (In V34, the **NewScreen** structure was used to set screen attributes so many of the tag options listed here have a corresponding flag in **NewScreen**.) In general, specifying a tag overrides the corresponding flag or field in the **NewScreen** structure if you supply one.

SA_ErrorCode

Extended error code. Data is a pointer to a long which will contain the error code on return if **OpenScreenTagList()** returns NULL. The error codes are described above.

SA_Left, SA_Top

Initial screen position (left edge and top edge). Data is a long, signed value. Offsets are relative to the text overscan rectangle.

If SA_Left is not specified and a **NewScreen** structure is not passed in the **OpenScreenTags/ TagList()** call and SA_Width is not specified or is specified as STDSCREENWIDTH, then the left edge of the screen will default to the left edge of the actual display clip of the screen. If the other conditions are met but some explicit SA_Width is specified, then the left edge defaults to zero (text overscan rectangle left edge). Likewise, the top edge may, independent of the left edge value, default to zero or to the top edge of the actual display clip. If SA_Top is not specified and a **NewScreen** structure is not passed in the **OpenScreenTags/TagList()** call and SA_Height is not specified or specified as STDSCREENHEIGHT, then the top edge of the screen will default to the top edge of the actual display clip of the screen. If the other conditions are met but some explicit SA_Height is specified, then the top edge defaults to zero (text overscan rectangle top edge). Prior to V36, left edge positioning is ignored and negative top edge positions are illegal.

When opening a full sized overscan screen, SA_Left should be set to the **MinX** value of the display clip **Rectangle** used for the screen and SA_Top should be set to the **MinY** value of the display clip. This may be taken from the defaults, as explained above, or explicitly set by the application. See the section below on "Overscan and the Display Clip" and the **OpenScreen()** Autodoc for more details.

If your screen is larger than your display clip, you may wish to set the SA_Left and SA_Top to values less than your display clip **MinX** and **MinY** in order to center a large screen on a smaller display. For an example of how to open a centered overscan screen, see module/screen.c in the IFF Appendix of the *Amiga ROM Kernel Reference Manual: Devices.*

SA_Width, SA_Height

Screen dimensions. Data is a long, unsigned value. These may be larger, smaller or the same as the dimensions of the display clip **Rectangle**. The use of STDSCREENWIDTH and STDSCREENHEIGHT will make the screen size equal to the display clip size.

To calculate the width of the display clip **Rectangle**, subtract the **MinX** value from the **MaxX** value plus one. Similarly, the height of the display clip may be calculated by subtracting the **MinY** value from the **MaxY** value plus one.

SA_Depth

Screen bitmap depth. Data is a long, unsigned value. The depth of the screen determines the number of available colors. See the "Graphics Primitives" for more information on hardware limitations of the display. Do not set the depth to a value greater than that supported by the specific display mode. This information is available to the application through the graphics library display database. The default is one bitplane.

SA_DisplayID

Extended display mode key for the screen. Data is a long, unsigned value. By using Release 2 **DisplayIDs** and the display database, applications can open a screen in any display mode available on a user's system, including PAL and NTSC modes. See the discussion of the display database in the "Graphics Primitives" chapter and the include file *<graphics/displayinfo.h>* for more information.

SA_Pens

Pen specification for the screen. Data is a pointer to a UWORD array terminated with `~0`, as found in the **DrawInfo** structure. Specifying the SA_Pens tag informs the system that the application is prepared to handle a screen rendered with the new 3D look of Intuition. See the section below on "DrawInfo and the 3D Look". Omitting this tag produces a screen with a flat look, but whose color usage is more backwards compatible.

SA_DetailPen

Detail pen for the screen. Data is a long, unsigned value. Used for rendering details in the screen title bar and menus. Use SA_Pens beginning with V36 for more control of pen specification. If SA_Pens is not specified, the screen will not get the new 3D look of Intuition available in Release 2. Instead this value will be used as the detail pen.

SA_BlockPen

Block pen for the screen. Data is a long, unsigned value. Used for rendering block fills in the screen title bar and menus. Use SA_Pens beginning with V36 for more control of pen specification. If SA_Pens is not specified, the screen will not get the new 3D look and this value will be used as the block pen.

SA_Title

Default screen title. Data is a pointer to a character string. This is the title displayed when the active window has no screen title or when no window is active on the screen.

SA_Colors

Specifies initial screen palette colors. Data is a pointer to an array of **ColorSpec** structures, terminated by a **ColorSpec** structure with **ColorIndex** = -1. Screen colors may be changed after the screen is opened with the graphics library functions **SetRGB4()** and **LoadRGB4()**. **ColorSpec** colors are right-justified, four bits per gun.

SA_FullPalette

Initialize color table to entire preferences palette (32 colors beginning with V36), rather than the subset from V34 and earlier, namely pens 0-3, 17-19, with remaining palette as returned by **GetColorMap()**. Data is a boolean value (use TRUE to set the flag). Defaults to FALSE.

SA_Font

Data is a pointer to a **TextAttr** structure (defined in *<graphics/text.h>*) which specifies the font, size and style to use for the screen. Equivalent to **NewScreen.Font**.

SA_SysFont

Alternative to SA_Font. Selects one of the preferences system fonts. Data is a long, unsigned value, with the following values defined:

0 Open screen with the user's preferred fixed width font (the default).
1 Open screen with the user's preferred font, which may be proportional.

The Workbench screen is opened with {SA_SysFont , 1}. Table 3-4 summarizes how the font selected at **OpenScreen()** time effects subsequent text operations in screens and windows.

Table 3-4: Intuition Font Selection Chart

What you tell **OpenScreen()**		**Screen** font	**Window.RPort** font
A.	NewScreen.Font = myfont	myfont	myfont
B.	NewScreen.Font = NULL	GfxBase->DefaultFont	GfxBase->DefaultFont
C.	{SA_Font, myfont}	myfont	myfont
D.	{SA_SysFont, 0}	GfxBase->DefaultFont	GfxBase->DefaultFont
E.	{SA_SysFont, 1}	Font Prefs Screen text	GfxBase->DefaultFont

Notes:

A and B are the options that existed in V34 and earlier OS versions.
C and D are tags in Release 2 equivalent to A and B respectively.
E is a new option for V36. The Workbench screen uses this option.
For 'myfont', any font may be used including a proportional one. This is true under all releases of the OS.
GfxBase->DefaultFont is always monospace. (This is the "System Default Text" from Font Preferences.)
Font Prefs Screen text (the "Screen Text" choice from Font Preferences) can be monospace or proportional.

The screen's font may not legally be changed after a screen is opened. The menu bar, window titles, menu items, and the contents of a string gadget all use the screen's font. The font used for menu items can be overridden in the menu item's **IntuiText** structure. Under V36 and higher, the font used in a string gadget can be overridden through the **StringExtend** structure. The font of the menu bar and window titles cannot be overridden.

The **Window.RPort** font shown above is the initial font that Intuition sets in your window's **RastPort**. It is legal to change that subsequently with **SetFont()**. **IntuiText** rendered into a window (either through **PrintIText()** or as a gadget's **GadgetText**) defaults to the window's **RastPort** font, but can be overridden using its **ITextFont** field. Text rendered with the graphics library call **Text()** uses the window's **RastPort** font.

SA_Type

Equivalent to the SCREENTYPE bits of the **NewScreen.Type** field. Data is a long, unsigned value which may be set to either CUSTOMSCREEN or PUBLICSCREEN (WBENCHSCREEN is reserved for system use). See the tags SA_BitMap, SA_Behind, SA_Quiet, SA_ShowTitle and SA_AutoScroll for the other attributes of the **NewScreen.Type** field.

SA_BitMap

Use a custom bitmap for this screen. Data is a pointer to a **BitMap** structure. This tag is equivalent to **NewScreen.CustomBitMap** and implies the CUSTOMBITMAP flag of the **NewScreen.Type** field. The application is responsible for allocating and freeing the screen's bitmap.

SA_Behind

Open this screen behind all other screens in the system. Data is a boolean value (TRUE to set flag). This tag is equivalent to the SCREENBEHIND flag of the **NewScreen.Type** field.

SA_Quiet

Disable Intuition rendering into screen. Data is a boolean value (TRUE to set flag). This tag is equivalent to the SCREENQUIET flag of the **NewScreen.Type** field. The screen will have no visible title bar or gadgets, but dragging and depth arrangement still function. In order to completely prevent Intuition from rendering into the screen, menu operations must be disabled for each window in the screen using WFLG_RMBTRAP.

SA_ShowTitle

Setting this flag places the screen's title bar in front of any backdrop windows that are opened on the screen. Data is a boolean value (TRUE to set flag). This tag is equivalent to the SHOWTITLE flag of the **NewScreen.Type** field. The title bar of the screen is always displayed behind any non-backdrop windows on that screen. This attribute can be changed after the screen is open with the **ShowTitle()** function.

SA_AutoScroll

Setting this flag will enable autoscroll for this screen when it is the active screen. (Currently, the screen may only be made active by activating a window in that screen either under user or application control.) Data is a boolean value (TRUE to set flag). This tag is equivalent to the AUTOSCROLL flag of the **NewScreen.Type** field.

Autoscroll means that screens larger than the visible display will automatically scroll when the user moves the mouse to the edge of the screen. Without this tag, the user moves the screen either by using the screen drag bar, or by pressing the mouse select button anywhere within the screen while holding down the left Amiga key and moving the mouse.

SA_PubName

Presence of this tag means that the screen is to be a public screen. Data is a pointer to a string. The string is the name of the public screen which is used by other applications to find the screen. This tag is order dependent, specify *before* SA_PubSig and SA_PubTask.

SA_PubSig, SA_PubTask

Task ID (returned by **FindTask()**) and signal for notification that the last window has closed on a public screen. Data for SA_PubSig is a long, unsigned value. Data for SA_PubTask is a pointer to a **Task** structure. These two tags are order dependent, and must be specified *after* the tag SA_PubName.

SA_Overscan

Set to one of the OSCAN_ specifiers to use a system standard overscan display clip and screen dimensions (unless otherwise specified). Data is a long, unsigned value. Do not specify this tag and SA_DClip. SA_Overscan is used to get one of the standard overscan dimensions, while SA_DClip is for custom dimensions. If a display clip is not specified with either SA_Overscan or SA_DClip, the display clip defaults to OSCAN_TEXT. See the section below on "Overscan and the Display Clip" for more information.

SA_DClip

Custom display clip specification. Data is a pointer to a **Rectangle** structure that defines the screen display clip region.

Public Screen Functions

Public screens are a new feature of Release 2 (V36). A public screen allows multiple applications to share a single screen thus saving memory. If your application opens a public screen, then other applications will be able to open their windows on your screen. In older versions of the operating system, only the Workbench screen could be shared so applications had to live within its limitations or use up Chip memory creating their own private, custom screens.

Now the system allows any screen to be set up as a public screen so there may be many public screens in memory at once, not just Workbench. This permits the power user to set up different work environments that multiple applications can share in a way that is memory efficient (each one with a display mode appropriate to a particular job).

Workbench is a special case public screen because it is the initial *default public screen*. The *default public screen* is the screen applications will get when they ask for a public screen but don't specify a name. Under normal conditions, Workbench is the default public screen and is created by the system at startup time. However, keep in mind that the default public screen can be changed (it's not always guaranteed to be Workbench).

> *Screens for the Novice.* If you're not sure what kind of screen to use, then use the default public screen. Under Release 2, you can open a window on the default public screen without doing any screen set-up work. See the "Intuition Windows" chapter for more details.

Generally, it is much easier to use an existing, public screen than to set up one of your own. Here are the basic functions you use to work with an existing public screen.

Table 3-5: Public Screen Functions

LockPubScreen()	Find Workbench or any other public screen; prevent it from closing while a window is opened or its attributes copied.
UnlockPubScreen()	Release the lock allowing the screen to later be closed.
SetDefaultPubScreen()	Establishes a given public screen as the default.
GetDefaultPubScreen()	Copies the name of the default screen to a user supplied buffer for use by the screen manager utility (the name is not needed by normal applications, use **LockPubScreen**(NULL) instead).
PubScreenStatus()	Converts a screen to private or public status.
SetPubScreenModes()	Controls the public screen global mode bits.

By using an existing public screen, an application is no longer responsible for setting up the display, however, it also loses flexibility and control. It can no longer set the palette or depth, and it cannot write directly into screen memory without cooperation from the owner of the public screen. (If these limitations are too confining, the application can create a new screen instead.)

ACCESSING A PUBLIC SCREEN BY NAME

The main calls for accessing an existing public screen are **LockPubScreen()** and **UnlockPubScreen()**. To use these functions you need to know the name of the public screen you want to access. If you do not know the name of the public screen or if you are not sure, you can lock the default public screen with **LockPubScreen**(NULL).

```
struct Screen *LockPubScreen( UBYTE * )
 VOID        UnlockPubScreen( UBYTE * , struct Screen *)
```

These calls enable the application to determine that a public screen exists, and to ensure its continued existence while opening a window on it. This function also serves as an improvement over the old **GetScreenData()** function from V34 by returning a pointer to the **Screen** structure of the locked screen so that its attributes can be examined.

Be sure to unlock the public screen when done with it. Note that once a window is open on the screen the program does not need to hold the screen lock, as the window acts as a lock on the screen. The pointer to the screen structure is valid as long as a lock on the screen is held by the application, or the application has a window open on the screen.

Locks should not be held without reason. Holding unnecessary locks on screens may prevent the user from closing a public screen that has no apparent activity. Keep in mind that as long as you have a window open on a public screen, the window acts as a lock preventing the screen from closing.

Shown here is a simple example of how to find the Workbench public screen using **LockPubScreen()** and **UnlockPubScreen()**.

```
;/* pubscreenbeep.c - Execute me to compile me with SAS 5.10
LC -b1 -cfistq -v -y -j73 pubscreenbeep.c
blink LIB:c.o pubscreenbeep.o TO pubscreenbeep LIB LIB:lc.lib LIB:amiga.lib
quit
*/

#include <exec/types.h>              /* Amiga data types.              */
#include <exec/libraries.h>
#include <intuition/intuition.h>     /* Lots of important Intuition    */
#include <intuition/screens.h>       /* structures we will be using.   */

#include <clib/exec_protos.h>        /* Function prototypes            */
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }   /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }   /* really */
#endif

struct Library *IntuitionBase;       /* Intuition library base          */

/* Simple example of how to find a public screen to work with in Release 2.
 */

VOID main(int argc, char **argv)
{
struct Screen  *my_wbscreen_ptr;    /* Pointer to the Workbench screen */

/* Open the library before you call any functions */
IntuitionBase = OpenLibrary("intuition.library",0);
if (NULL != IntuitionBase)
   {
   if(IntuitionBase->lib_Version>=36)
      {
      /* OK, we have the right version of the OS so we can use
      ** the new public screen functions of Release 2 (V36)
      */
```

```
        if(NULL!=(my_wbscreen_ptr=LockPubScreen("Workbench")))
            {
            /* OK found the Workbench screen.                    */
            /* Normally the program would be here.  A window could */
            /* be opened or the attributes of the screen copied    */
            DisplayBeep(my_wbscreen_ptr);

            UnlockPubScreen(NULL,my_wbscreen_ptr);
            }
        }
    else
        {
        /* Prior to Release 2 (V36), there were no public screens,     */
        /* just Workbench.  In those older systems, windows can be     */
        /* opened on Workbench without locking or a pointer by setting */
        /* the Type=WBENCHSCREEN in struct NewWindow.  Attributes can  */
        /* be obtained by setting the Type argument to WBENCHSCREEN in */
        /* the call to GetScreenData().                                */
        }
    CloseLibrary(IntuitionBase);
    }
}
```

THE DEFAULT PUBLIC SCREEN AND WORKBENCH

As mentioned earlier, Workbench is a special case public screen because it is the initial *default public screen*. There are other reasons Workbench has a special status. Normally, it's the first thing the user sees because it is the default user interface on all Amiga computers. Many older applications written for V34 and earlier versions of the OS expect to run in the Workbench screen. Also, Workbench is currently the only public screen supported by system Preferences and the only screen Intuition can automatically open.

Because of its close ties with the operating system, there are some extra functions available to manipulate the Workbench screen. One function which controls both Workbench and other public screens is **SetPubScreenModes()**. This function controls the global public screen mode bits, SHANGHAI and POPPUBSCREEN. If the SHANGHAI mode bit is set, older applications which expect to open on the Workbench screen will open instead on the default public screen (which may or may not be the Workbench screen). The POPPUBSCREEN bit controls whether public screens will be popped to the front when a window is opened. These modes are documented in the ''Intuition Windows'' chapter in the section on ''Windows and Screens''.

Other functions which control the Workbench screen are listed in the table below.

Table 3-6: Workbench Public Screen Functions

WBenchToBack()	Move the Workbench screen behind all other screens.
WBenchToFront()	Move the Workbench screen in front of all other screens.
OpenWorkBench()	Open the Workbench screen. If the screen is already open, this call has no effect. This call will re-awaken the Workbench application if it was active when **CloseWorkBench()** was called.
CloseWorkBench()	Attempt to reclaim memory used for the Workbench screen. If successful, this call closes the screen and puts the Workbench application to sleep. This call fails if any application has windows open or locks on the Workbench screen.

Programs can attempt to reclaim memory used by the Workbench screen by calling **CloseWorkBench()**. Programs that have closed Workbench, should call **OpenWorkBench()** as they exit or allow the user to re-open the screen through a menu or gadget.

If Workbench is closed, any of the following events can re-open it: calling **OpenWorkBench()**; opening a window on the Workbench (including **EasyRequests()** such as the DOS ''Insert Disk'' requester); calling **LockPubScreen**("Workbench"); calling **LockPubScreen**(NULL) when Workbench is the default public screen.

TAKING A NEW CUSTOM SCREEN PUBLIC

Applications that open a new screen should consider taking the screen public. If the screen's characteristics are not very esoteric, making the screen public is useful because it allows other applications to share the working context. This makes an application more powerful and more attractive to the user because it allows the user to add supporting applications and utilities from other vendors to make a customized and integrated work environment.

To make your own custom screen into a public screen that other applications may use, you give the screen a public name and then register the screen with Intuition. The screen must be declared as public in the **OpenScreenTagList()** call by specifying a public name string with the SA_PubName tag. The application's task ID and a signal bit may also be registered when the screen is opened with the SA_PubTask and SA_PubSig tags. If these tags are given, the system will signal your task when the last window on the screen closes.

When a new public screen is opened, it starts out private so the application can perform any desired initialization (for instance, opening a backdrop window) before the screen is made public. Use the **PubScreenStatus()** function to make the screen public and available to other applications (or to take the screen private again, later). The screen may not be taken private or closed until all windows on the screen are closed and all locks on the screen are released. However, the screen does not need to be made private before closing it.

CloseScreen() will fail if an attempt is made to close a public screen that still has visitor windows or locks on it. If the user selects close screen, but the screen will not close due to visitor windows, a requester should be displayed informing the user of the condition and instructing them to close any windows before closing the screen.

SEARCHING THE PUBLIC SCREEN LIST

To access an existing public screen the application may take one of three approaches. To get a lock on the default public screen, either **LockPublicScreen**(NULL) or {WA_PubScreenName , NULL} may be used.

If the name of the screen is known, the application may use **LockPubScreen(Name)** to gain a lock on the screen as shown in the example above (or use **OpenWindowTagList()** with the **WA_PubScreenName** tag as described in the ''Intuition Windows'' chapter). Failure to lock the screen or open the window probably indicates that the screen does not exist.

A third approach is to search the public screen list for a screen that meets the requirements of the application. These requirements may be related to the name or attributes of the screen. Here are the functions to use with the public screen list maintained by Intuition.

Table 3-7: Public Screen List Functions

LockPubScreenList()	Lock the public screen list maintained by Intuition so that it may be quickly copied
UnlockPubScreenList()	Release the lock on the public screen list
NextPubScreen()	Find the next screen in the public screen list

The main function used to access the public screen list is **LockPubScreenList()**. This function, intended for use by the public screen manager utility, locks the list to allow data from it to be quickly copied. The list is stored in an Exec **List** structure, with each node in the list being a **PubScreenNode** structure. See *<intuition/screens.h>* for details.

Do not interpret the list while in a locked state, instead, copy any values required to local variables and release the lock. All required data must be copied, including the name of the screen which is not part of the structure. Pointers that reference the list or structures attached to the list are not valid after releasing the lock. Once the lock is released, the screen pointers in the list (**psn_Screen**) may be tested for equality against other screen pointers, but referencing any part of the screen structure from this pointer is strictly illegal. After the lock is released with **UnlockPubScreenList()**, the application may access the data in the screen structure by obtaining a lock on the screen using **LockPubScreen()** with the name of the screen.

The application should only require accessing three fields in the **PubScreenNode**, these are **ln_Name**, **psn_Screen** and **psn_Flags**. The name of the public screen is maintained in the **ln_Name** field of the **Node** (**psn_Node**) structure. Access to other information on the screen may be done by getting a lock on this name and reading the data from the **Screen** structure. The screen pointer (**psn_Screen**) may only be used for testing against other screen pointers, never reference the screen structure from this value. Finally, the public screen flags are maintained in **psn_Flags**. Currently, only PSNF_PRIVATE is defined for this field. PSNF_PRIVATE indicates that the screen is not currently public.

Remember that all information in the public screen list is transitory, that is, it may change at any time. Do not rely on the values in the list. The only way to ensure the existence or mode of a screen is to lock it, either directly with **LockPubScreen()** or by opening a window on the screen. To update the copy of the list, lock it and copy the data again. Don't forget to release the lock when finished.

As an alternative to dealing with the public screen list, **NextPubScreen()** can be used. This call takes the name of a public screen as its argument and returns the name of the next screen in the public screen list. This helps an application move a window through the entire rotation of public screens. Repeated calls to **NextPubScreen()** could be used to get the names of all public screens one at a time. Keep in mind though that the list of public screens is subject to sudden change; the task that owns a public screen might close it after you obtain the name, but before you access the screen.

Always use **LockPubScreen()** to access screen information after scanning the public screen list.

DrawInfo and the 3D Look

In Release 2, whenever a new screen is created, Intuition also creates an auxiliary data structure called a **DrawInfo**. The **DrawInfo** structure provides information Intuition uses to support the new 3D look of Release 2 and specifies graphical information for applications that use the screen. The information includes such items as aspect ratio (resolution), font, number of colors and drawing pens.

```
struct DrawInfo
{
    UWORD       dri_Version;    /* will be  DRI_VERSION                 */
    UWORD       dri_NumPens;    /* guaranteed to be >= numDrIPens       */
    UWORD       *dri_Pens;      /* pointer to pen array                    */

    struct TextFont     *dri_Font;  /* screen default font              */
    UWORD       dri_Depth;          /* (initial) depth of screen bitmap */

    struct {    /* from DisplayInfo database for initial display mode   */
      UWORD   X;
      UWORD   Y;
    }           dri_Resolution;

    ULONG       dri_Flags;            /* defined below                  */
    ULONG       dri_Reserved[7];      /* avoid recompilation ;^)        */
};
```

Before an application uses fields in the **DrawInfo** structure, it should check the version of the structure to ensure that all fields are available. If the field **dri_Version** is greater than or equal to the constant DRI_VERSION that the application was compiled with, it can be assured that all fields in **DrawInfo** that it knows about are being supported by Intuition.

THE PEN SPECIFICATION IN DRAWINFO

The drawing pen specification in **DrawInfo.dri_Pens** allows applications to use appropriate colors for graphic operations such as drawing text, shading 3D objects and filling items selected by the user.

Intuition has two default sets of pens, one for multi-bitplane screens and one for single bitplane screens. In addition, there is a special compatibility mode for screens that do not specify the SA_Pens tag.

New 3D Look

The is the full 3D look as found by default on the Workbench screen in Release 2. Objects are drawn so that light appears to come from the upper left of the screen with shadows cast to the lower right giving them a three-dimensional look.

Monochrome New Look

It is impossible to produce the full 3D look in a single bitplane (two color) screen. Intuition provides a fallback pen specification that is used in monochrome screens with no loss of information.

Compatible New Look

Custom screens that do not provide the SA_Pens tag are assumed to have no knowledge of the pen array. They are rendered in a special version of the monochrome new look, which uses the screen's **DetailPen** and **BlockPen** to get its colors. This is provided for compatibility with V34 and older versions of the operating system.

It is very easy for an application to use the default pen specification. Simply specify an empty pen specification (in C, `{~0}`), and Intuition will fill in all of the values with defaults appropriate for the screen. This technique is demonstrated in the first two examples listed earlier in this chapter.

For certain applications, a custom pen specification is required. A custom pen specification is set up when the screen is opened by using the SA_Pens tag and a pointer to a pen array. Currently, Intuition uses nine pens to support the 3D look. The application can specify all of these, or only a few pens and Intuition will fill in the rest. Intuition will only fill in pens that are past the end of those specified by the application, there is no facility for using default values for "leading" pens (those at the beginning of the array) without using the defaults for the rest of the pens.

Using the pen specification of an existing public screen is a bit more involved. First, the application must get a pointer to the screen structure of the public screen using the **LockPubScreen()** call. A copy of the screen's **DrawInfo** structure may then be obtained by calling **GetScreenDrawInfo()**. The **DrawInfo** structure contains a copy of the pen specification for the screen that can be used in the **OpenScreenTagList()** call with the SA_Pens tag. The pen array is copied to the data structures of the new screen (it is not kept as a pointer to the information passed), so the application may immediately call **FreeScreenDrawInfo()** and **UnlockPubScreen()** after the new screen is open.

```
/* publicscreen.c
** open a screen with the pens from a public screen.
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y publicscreen
** blink FROM LIB:c.o publicscreen.o TO publicscreen LIB LIB:lc.lib LIB:amiga.lib
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

VOID usePubScreenPens(void);

struct Library *IntuitionBase;

/* main(): open libraries, clean up when done.
*/
VOID main(int argc, char **argv)
{
IntuitionBase = OpenLibrary("intuition.library",0);
if ( IntuitionBase != NULL )
    {
    /* Check the version number; Release 2 is */
    /* required for public screen functions   */
    if (IntuitionBase->lib_Version >= 37)
        {
        usePubScreenPens();
        }
    CloseLibrary(IntuitionBase);
    }
}

/* Open a screen that uses the pens of an existing public screen
** (the Workbench screen in this case).
*/
VOID usePubScreenPens(void)
{
struct Screen *my_screen;
struct TagItem screen_tags[2];
UBYTE *pubScreenName = "Workbench";

struct Screen *pub_screen = NULL;
struct DrawInfo *screen_drawinfo = NULL;
```

```
/* Get a lock on the Workbench screen */
pub_screen = LockPubScreen(pubScreenName);
if ( pub_screen != NULL )
    {
    /* get the DrawInfo structure from the locked screen */
    screen_drawinfo = GetScreenDrawInfo(pub_screen);
    if ( screen_drawinfo != NULL)
        {
        /* the pens are copied in the OpenScreenTagList() call,
        ** so we can simply use a pointer to the pens in the tag list.
        **
        ** This works better if the depth and colors of the new screen
        ** matches that of the public screen.  Here we are forcing the
        ** workbench screen pens on a monochrome screen (which may not
        ** be a good idea).  You could add the tag:
        **      (SA_Depth, screen_drawinfo->dri_Depth)
        */
        screen_tags[0].ti_Tag  = SA_Pens;
        screen_tags[0].ti_Data = (ULONG)(screen_drawinfo->dri_Pens);
        screen_tags[0].ti_Tag  = TAG_END;
        screen_tags[0].ti_Data = NULL;

        my_screen = OpenScreenTagList(NULL, screen_tags);
        if (my_screen != NULL)
            {
            /* We no longer need to hold the lock on the public screen
            ** or a copy of its DrawInfo structure as we now have our
            ** own screen.  Release the screen.
            */
            FreeScreenDrawInfo(pub_screen,screen_drawinfo);
            screen_drawinfo = NULL;
            UnlockPubScreen(pubScreenName,pub_screen);
            pub_screen = NULL;

            Delay(90);   /* should be rest_of_program */

            CloseScreen(my_screen);
            }
        }
    }

/* These are freed in the main loop if OpenScreenTagList() does
** not fail.  If something goes wrong, free them here.
*/
if ( screen_drawinfo != NULL )
    FreeScreenDrawInfo(pub_screen,screen_drawinfo);
if ( pub_screen!= NULL )
    UnlockPubScreen(pubScreenName,pub_screen);
}
```

Beginning with V36, the pen specification for the Workbench screen happens to match the Intuition default specification, however, this is not required and may change in the future. To create a screen that uses the pens defined in the Workbench screen, the application must get a copy of the pen array from the Workbench screen and use this copy with the SA_Pens tag as described above.

Here is a list of the pens defined under V36 that support the 3D look along with their uses. To read the value of a particular pen, use **UWORD penvalue = myDrawInfo->dri_Pens[PENNAME]**, where **myDrawInfo** is a pointer to a **DrawInfo** structure and **PENNAME** is taken from the list below:

DETAILPEN
: Pen compatible with V34. Used to render text in the screen's title bar.

BLOCKPEN
: Pen compatible with V34. Used to fill the screen's title bar.

TEXTPEN
: Pen for regular text on BACKGROUNDPEN.

SHINEPEN
    Pen for the bright edge on 3D objects.

SHADOWPEN
    Pen for the dark edge on 3D objects.

FILLPEN
    Pen for filling the active window borders and selected gadgets.

FILLTEXTPEN
    Pen for text rendered over FILLPEN.

BACKGROUNDPEN
    Pen for the background color. Currently must be zero.

HIGHLIGHTTEXTPEN
    Pen for "special color" or highlighted text on BACKGROUNDPEN.

THE FONT SPECIFICATION IN DRAWINFO

Font information for a screen comes from a number of different places.

SA_Font
The application may specify the font to be used in a screen by providing the SA_Font tag with a **TextAttr** structure. In this case, the font will be used by the screen and will be the default font for the **RastPort** of any window opening in the screen.

SA_SysFont, 0
If the application requests the user's preferred monospace font, it is taken from **GfxBase->DefaultFont**. Any window's **RastPorts** are also initialized to use this same font.

SA_SysFont, 1
The screen font selected by the user from the Preferences font editor may be used for the screen by using the SA_SysFont tag. This font, the "preferred screen font", may be proportional. For compatibility reasons, if this font is specified for the screen, the window's **RastPort** will be initialized to **GfxBase->DefaultFont** (a non-proportional font).

To access information on an open screen's font, the application may reference **Screen.Font** or **DrawInfo.dri_Font**. These fonts are identical, the **DrawInfo** structure simply provides an alternate method of accessing the information. Note that **Screen.Font** is a pointer to a **TextAttr** structure and that **DrawInfo.dri_Font** is a pointer to a **TextFont** structure. The application may use whichever form is best suited to its requirements.

It is illegal to change the screen's font after the screen is opened. This means that the font specified in the **Screen** and **DrawInfo** structures is guaranteed to remain open as long is the screen is open.

The menu bar, window titles, menu items, and the contents of a string gadget all use the screen's font. The font used for menu items can be overridden in the menu item's **IntuiText** structure. Under V36 and higher, the font used in a string gadget can be overridden through the **StringExtend** structure. The font of the menu bar and window titles cannot be overridden.

For more information on screen fonts, see the description of the SA_Font and SA_SysFont tags in the "Screen Attributes" section above.

CLONING A PUBLIC SCREEN (WORKBENCH)

User preferences for screen attributes are generally reflected in the Workbench screen or in the default public screen. In some cases it may be useful to create a new screen with the same attributes.

Under V34, information on a screen was available through the **GetScreenData()** call. Due to extensions in V36 screen and graphics capabilities, this call is no longer sufficient to completely describe the display. Applications should now use a variety of calls; the specific call depends on the information required.

LockPubScreen() returns a pointer to the **Screen** structure of a specific screen. **GetScreenDrawInfo()** returns rendering information on the screen, such as the pen array and font used. **QueryOverscan()** returns the overscan information of a specific display mode (for more information, see the section on "Overscan and the Display Clip").

The example below shows how to use **GetScreenDrawInfo()** to examine the attributes of the Workbench screen so that a new screen with the same attributes can be created.

```
struct DrawInfo *GetScreenDrawInfo( struct Screen * )
```

The attributes required to clone an existing screen are its width, height, depth, pens and mode. The pens and screen depth are available through the **DrawInfo** structure. The width and height may be obtained from the **Screen** structure. (The width and height may be larger than the overscan area if the screen is scrollable, and autoscroll may always be enabled as it does not effect displays smaller than or equal to the overscan area.)

The screen's display mode can be obtained using the graphics library call **GetVPModeID()**. This call returns the display ID of an existing screen which can then be used as the data for the SA_DisplayID tag in **OpenScreenTagList()**. Note that the example assumes the screen should be open to the user's text overscan preference. If an exact copy of the display clip of the existing screen is required, use the **VideoControl()** command of the graphics library to access the **ViewPortExtra** structure.

The colors of the screen may be copied using the graphics library calls **GetRGB4()**, **SetRGB4()**, **SetRGB4CM()** and **LoadRGB4()**. The example code does not copy the colors.

The example copies the font from the cloned screen. A reasonable alternative would be to use the user's preference font, which may be accessed through the SA_SysFont tag.

```
/* clonescreen.c
** clone an existing public screen.
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y clonescreen
** blink FROM LIB:c.o clonescreen.o TO clonescreen LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

#include <string.h>
```

```
#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

VOID cloneScreen( UBYTE * );

struct Library *IntuitionBase;
struct GfxBase *GfxBase;

/*
** Open all libraries for the cloneScreen() subroutine.
*/
VOID main(int argc, char **argv)
{
UBYTE *pub_screen_name = "Workbench";

IntuitionBase = OpenLibrary("intuition.library",0);
if (IntuitionBase != NULL)
    {
    /* Require version 37 of Intuition. */
    if (IntuitionBase->lib_Version >= 37)
        {
        /* Note the two methods of getting the library version
        ** that you really want.
        */
        GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",37);
        if (GfxBase != NULL)
            {
            cloneScreen(pub_screen_name);

            CloseLibrary((struct Library *)GfxBase);
            }
        }
    CloseLibrary(IntuitionBase);
    }
}

/* Clone a public screen whose name is passed to the routine.
**    Width, Height, Depth, Pens, Font and DisplayID attributes are
** all copied from the screen.
**    Overscan is assumed to be OSCAN_TEXT, as there is no easy way to
** find the overscan type of an existing screen.
**    AutoScroll is turned on, as it does not hurt.  Screens that are
** smaller than the display clip will not scroll.
*/

VOID cloneScreen(UBYTE *pub_screen_name)
{
struct Screen *my_screen;
ULONG  screen_modeID;
UBYTE *pub_scr_font_name;
UBYTE *font_name;
ULONG  font_name_size;
struct TextAttr pub_screen_font;
struct TextFont *opened_font;

struct Screen   *pub_screen = NULL;
struct DrawInfo *screen_drawinfo = NULL;

/* name is a (UBYTE *) pointer to the name of the public screen to clone */
pub_screen = LockPubScreen(pub_screen_name);
if (pub_screen != NULL)
    {
    /* Get the DrawInfo structure from the locked screen
    ** This returns pen, depth and font info.
    */
    screen_drawinfo = GetScreenDrawInfo(pub_screen);
    if (screen_drawinfo != NULL)
        {
        screen_modeID = GetVPModeID(&(pub_screen->ViewPort));
        if( screen_modeID != INVALID_ID )
            {
            /* Get a copy of the font
            ** The name of the font must be copied as the public screen may
            ** go away at any time after we unlock it.
```

```
        ** Allocate enough memory to copy the font name, create a
        ** TextAttr that matches the font, and open the font.
        */
        pub_scr_font_name = screen_drawinfo->dri_Font->tf_Message.mn_Node.ln_Name;
        font_name_size = 1 + strlen(pub_scr_font_name);
        font_name = AllocMem(font_name_size, MEMF_CLEAR);
        if (font_name != NULL)
            {
            strcpy(font_name, pub_scr_font_name);
            pub_screen_font.ta_Name  = font_name;
            pub_screen_font.ta_YSize = screen_drawinfo->dri_Font->tf_YSize;
            pub_screen_font.ta_Style = screen_drawinfo->dri_Font->tf_Style;
            pub_screen_font.ta_Flags = screen_drawinfo->dri_Font->tf_Flags;

            opened_font = OpenFont(&pub_screen_font);
            if (opened_font != NULL)
                {
                /* screen_modeID may now be used in a call to
                ** OpenScreenTagList() with the tag SA_DisplayID.
                */
                my_screen = OpenScreenTags(NULL,
                    SA_Width,      pub_screen->Width,
                    SA_Height,     pub_screen->Height,
                    SA_Depth,      screen_drawinfo->dri_Depth,
                    SA_Overscan,   OSCAN_TEXT,
                    SA_AutoScroll, TRUE,
                    SA_Pens,       (ULONG)(screen_drawinfo->dri_Pens),
                    SA_Font,       (ULONG)&pub_screen_font,
                    SA_DisplayID,  screen_modeID,
                    SA_Title,      "Cloned Screen",
                    TAG_END);
                if (my_screen != NULL)
                    {
                    /* Free the drawinfo and public screen as we don't
                    ** need them any more.  We now have our own screen.
                    */
                    FreeScreenDrawInfo(pub_screen,screen_drawinfo);
                    screen_drawinfo = NULL;
                    UnlockPubScreen(pub_screen_name,pub_screen);
                    pub_screen = NULL;

                    Delay(300);   /* should be rest_of_program */

                    CloseScreen(my_screen);
                    }
                CloseFont(opened_font);
                }
            FreeMem(font_name, font_name_size);
            }
        }
    }
}

/* These are freed in the main loop if OpenScreenTagList() does
** not fail.  If something goes wrong, free them here.
*/
if (screen_drawinfo != NULL )
    FreeScreenDrawInfo(pub_screen,screen_drawinfo);
if (pub_screen != NULL )
    UnlockPubScreen(pub_screen_name,pub_screen);
}
```

Overscan and the Display Clip

Screens may be larger or smaller than the defined display area (*overscan rectangle* or *display clip*). When a screen is smaller than the display area, the display clip acts as a "container" for the screen. The screen may be moved anywhere within the display clip. When a screen is larger than the display area, the display clip acts as a "window" into the screen. The screen may be moved so that different parts are visible. Each dimension of the screen is independent and may be larger than, the same as, or smaller than the dimensions of the display clip.

The system is very flexible in its specification of screen size. Unless an application fixes its screen size with hard coded values, it should be prepared to handle the possibility that the user has changed the default overscan presets or the default monitor (NTSC/PAL).

Use the constants STDSCREENHEIGHT and STDSCREENWIDTH with the SA_Width and SA_Height tags to open a screen the same size as the display clip. These constants will work with any of the preset overscan values set with SA_Overscan, and with custom overscan values set with SA_DClip.

PRESET OVERSCAN VALUES

Four preset overscan dimensions are provided. Applications that support overscan should use these preset values where possible since they will be tailored to each individual system. Avoid using custom values that happen to look good on a specific system. However, be aware that the size and positioning of overscan screens can be different on every system depending on how the user has set Overscan Preferences. These preset values are also dependent on the underlying display mode so keep in mind that both offset and size parameters will change under different screen modes. Overscan presets can be used, among other things, with the SA_Overscan tag to set the size of the screen's display clip or passed as an argument to **QueryOverscan()** to find their current overscan settings.

OSCAN_TEXT

This overscan region is based on user preference settings and indicates a display that is completely within the visible bounds of the monitor. The **View** origin is set to the upper left corner of the text overscan rectangle which is the highest leftmost point known to be visible on the physical display. This position is set by the user through the Overscan Preferences editor. All screen positions and display clips are relative to this origin.

OSCAN_STANDARD

The edges of OSCAN_STANDARD display are also based on user preferences and are set to be just outside the visible bounds of the monitor. OSCAN_STANDARD provides the smallest possible display that will fill the entire screen with no border around it. Parts of the display created with OSCAN_STANDARD may not be visible to the user.

OSCAN_MAX

Create the largest display fully supported by Intuition and the graphics library. This is the largest size for which all enclosed sizes and positions are legal. Parts of the display created with OSCAN_MAX may not be visible to the user.

OSCAN_VIDEO

Create the largest display, restricted by the hardware. This is the only legal size and position that is possibly (but not necessarily) larger than OSCAN_MAX. You must use the exact size and position specified. OSCAN_VIDEO does not support variable left edge, top edge positioning. Parts of the display created with OSCAN_VIDEO may not be visible to the user.

If custom clipping is required, a display clip may be explicitly specified using the SA_DClip tag and a **Rectangle** structure specification. This custom rectangle must fit within the OSCAN_MAX rectangle, offset included. It is not permitted to specify custom rectangles whose values are in between OSCAN_MAX and OSCAN_VIDEO, nor is it permitted to specify rectangles larger than OSCAN_VIDEO. For an example of how to open a centered overscan screen based on user preferences, see the module/screen.c listing in the IFF Appendix of the *Amiga ROM Kernel Reference Manual: Devices.*

Use the Graphics library call **VideoControl()** to find the true display clip of a screen. See the Graphics Autodocs and the chapter "Graphics Primitives" for more information on **VideoControl()**. The **ViewPortExtra** structure contains the display clip information.

If any dimension of a screen is not equal to the equivalent display clip dimension, then the screen may be scrolled. If the screen's dimensions are smaller than the display clip, then the screen may be positioned within the display clip. If the screen is larger than the display clip, then it may be positioned such that any part of the screen is visible.

AutoScroll may be activated by setting the tag SA_AutoScroll. Screens will only scroll when they are the active screen. Activate a window in the screen to make the screen active.

> *About the Default Display Clip.* The default display clip for a screen is the entire screen, that is, the rectangle starting from the upper left corner of the screen and ending at the lower right corner of the screen. This display clip is only used if the application does not specify SA_Overscan or SA_DClip. When using this default display clip the screen will not scroll as the screen exactly fits into the clipping region.

When opening a window in an overscanned screen, it is often useful to open it relative to the visible part of the screen rather than relative to the entire screen. Use **QueryOverscan()** to find the overscan region and where the screen is positioned relative to it.

```
LONG QueryOverscan(ULONG displayID, struct Rectangle *rect, WORD overscanType )
```

This example was taken from the chapter "Intuition Windows" in the section "Visible Display Sized Window Example". The complete example is reproduced there.

```
/* this technique returns the text overscan rectangle of the screen that we
** are opening on.  If you really need the actual value set into the display
** clip of the screen, use the VideoControl() command of the graphics library
** to return a copy of the ViewPortExtra structure.  See the Graphics
** library chapter and Autodocs for more details.
**
** GetVPModeID() is a graphics call...
*/

screen_modeID = GetVPModeID(&(pub_screen->ViewPort)))
if (screen_modeID != INVALID_ID)
    {
    if ( QueryOverscan(screen_modeID, &rect, OSCAN_TEXT) )
        {
        /* if this screen's origin is up or to the left of the */
        /* view origin then move the window down and to the right */
        left = max(0, -pub_screen->LeftEdge);
        top  = max(0, -pub_screen->TopEdge);

        /* get width and height from size of display clip */
        width  = rect.MaxX - rect.MinX + 1;
        height = rect.MaxY - rect.MinY + 1;

        /* adjust height for pulled-down screen (only show visible part) */
        if (pub_screen->TopEdge > 0)
            height -= pub_screen->TopEdge;

        /* ensure that window fits on screen */
        height = min(height, pub_screen->Height);
        width  = min(width,  pub_screen->Width);

        /* make sure window is at least minimum size */
        width  = max(width,  MIN_WINDOW_WIDTH);
        height = max(height, MIN_WINDOW_HEIGHT);
        }
    }
```

Intuition Screens and the Graphics Library

As previously mentioned, an Intuition screen is related to a number of underlying graphics library structures.

Table 3-8: Graphics Data Structures Used with Screens

Structure Name	Description	Defined in Include File
View	Root structure of the graphics display system	*<graphics/view.h>*
ViewPort	The graphics structure that corresponds to a screen	*<graphics/view.h>*
BitMap	Contains size and pointers to the screen's bit planes	*<graphics/gfx.h>*
ColorMap	Contains size and pointer to the screen's color table	*<graphics/view.h>*
RastPort	Holds drawing, pen and font settings and the **BitMap** address	*<graphics/rastport.h>*

These data structures are unified in Intuition's **Screen** structure (which also incorporates higher level Intuition constructs such as menus and windows). Here's a brief explanation of the graphics library structures used with Intuition.

View

The **View** is the graphics structure that corresponds to the whole display, including all visible screens. The system has just one **View**; it's what you see on the monitor. The address of the **View** may be obtained from any screen by using **ViewAddress()**.

ViewPort

The **ViewPort** is the underlying graphics structure corresponding to a screen. Every screen has one **ViewPort**. To get the address of the **ViewPort** from the **Screen** structure, use **(&my_screen->ViewPort)**. From the **ViewPort** an application may obtain pointers to all the screen's bitplanes and to its color table.

BitMap

The **BitMap** structure contains pointers to all the bit planes (up to 8) and their sizes. For future compatibility, use **(my_screen->RastPort.BitMap)** to get the address of the **BitMap** from the screen rather than **(&my_screen->BitMap)**.

The **BitMap.BytesPerRow** field specifies the number of bytes that have been allocated for each raster line. This may be larger than the screen width depending on display alignment restrictions. Alignment restrictions may change. Always use this variable, not a hard-coded value.

ColorMap

The **ColorMap** contains a pointer to the color table, an array of 32 WORDs for the hardware color registers. Use **SetRGB4()**, **GetRGB4()**, **SetRGB4CM()** and **LoadRGB4()** from the graphics library to access the color table. Do not read or write it directly.

RastPort

A **RastPort** controls the graphics rendering to *any* display area (not just screens). Screens have a **RastPort** to allow direct rendering into the screen. Applications may find the **RastPort** address of a screen with **(&my_screen->RastPort)**. This generally is not useful since applications normally render into windows.

CHANGING SCREEN COLORS

Screen colors are set at the time the screen is opened with the SA_Colors tag. If the colors need to be changed after the screen is opened, the graphics library function, **LoadRGB4()** should be used. To change a single entry in the color table, use **SetRGB4()** and **SetRGB4CM()**. See the "Graphics Primitives" chapter for more information on these functions.

DIRECT SCREEN ACCESS

Sometimes an application may want direct access to the custom screen's bitmap to use with low-level graphics library calls. This may be useful if the application needs to do custom manipulation of the display but also needs Intuition functionality. For instance, an application may want to use the graphics library primitives to perform double buffering then, when detecting user input, switch to Intuition control of the screen so that windows, gadgets and menus may be used to process the user input. If an application chooses to combine these techniques, it must take special care to avoid conflicts with Intuition rendered graphics. An example of how to do this is listed in the next section, "Advanced Screen Programming".

Application programs that open custom screens may use the screen's display memory in any way they choose. However, this memory is also used by Intuition for windows and other high level display components on the screen. Writing directly to the screen memory, whether through direct access or through graphics library calls that access the screen's **RastPort**, is not compatible with many Intuition constructs such as windows and menus.

Techniques such as this require great care and understanding of the Amiga. If possible, the application should avoid these techniques and only use standard Intuition display and input processing. Directly accessing the screen's bitmap, while possible, is not recommended. A better way to access the screen display is through windows. Windows provide access to the screen through layers which perform clipping and arbitration between multiple drawing areas.

Alternatives to writing directly to a screen, such as using a backdrop window, greatly limit the number of cases where an application must access screen memory. The **ShowTitle()** function allows the screen's title bar layer to be positioned in front of or behind any backdrop windows that are opened on the screen. Hence, a backdrop window may be created that uses the entire visible area of the monitor. Application programs that use existing public screens do not have the same freedom to access the screen's display memory as they do with custom screens. In general, public screens must be shared through the use of windows and menus rather than directly accessing the screen's display memory.

> *Use Direct Access Only On Screens You Own.* An application may not steal the bitmap of a screen that it does not own. Stealing the Workbench screen's bitmap, or that of any other public screen, is strictly illegal. Accessing the underlying graphics structures of a screen may only be done on custom screens opened by the application itself.

> *Do Not Perform Layers Operations Directly.* While layers are not part of the graphics library, it is appropriate to mention them here. Certain types of layers operations are not allowed with Intuition. You may not, for example, call **SizeLayer()** on a window (use **SizeWindow()** instead). To access layers library features with screens, use Intuition windows!

A custom screen may be created to allow for modification of the screen's Copper list. The Copper is the display synchronized co-processor that handles the actual video display by directly affecting the hardware registers. See the *Amiga Hardware Reference Manual* or the graphics library chapters for more information on programming the Copper.

SCREEN FUNCTIONS THAT INTEGRATE INTUITION AND GRAPHICS

These functions, normally used only by the system, integrate high-level Intuition structures with the lower-level constructs used by the graphics library to create the display.

Table 3-9: Screen Functions That Integrate Intuition and Graphics

MakeScreen()	Update a single screen's copper list
RethinkDisplay()	Merge copper lists from all screens to form a **View**
RemakeDisplay()	Update all screen copper lists then merge them to form a **View**

Advanced Intuition programmers may use these functions to achieve special screen effects such as double-buffering or dual-playfield Intuition screens. For examples of these see the next section.

MakeScreen() updates, but does not install, a screen's Copper list. This function is the Intuition equivalent of the low-level **MakeVPort()** graphics library function. **MakeScreen()** performs the **MakeVPort()** call, synchronized with Intuition's own use of the screen's **ViewPort**. Call **RethinkDisplay()** after **MakeScreen()** to allow the new Copper list for the screen to take effect. The **MakeScreen()** function takes one argument, a pointer to the **Screen** that contains the Copper list to be updated.

RethinkDisplay() combines all the screen's copper lists into a single view. This procedure performs the Intuition global display reconstruction, which includes massaging some of Intuition's internal state data, rethinking all of the Intuition screen **ViewPorts** and their relationship to one another, and, finally, reconstructing the entire display by merging the new screens into the graphics **View** structure. Custom screens that handle their own Copper instructions, use this call to install the Copper list previously updated with **MakeScreen()**. **RethinkDisplay()** calls lower-level graphics primitives **MrgCop()** and **LoadView()** to install the Copper list. This function takes no arguments.

RemakeDisplay() remakes the entire Intuition display. It is equivalent to calling **MakeScreen()** for each screen in the system, then calling **RethinkDisplay()**. This routine performs a **MakeVPort()** (graphics primitive) on every Intuition screen and then calls **RethinkDisplay()** to recreate the **View**. It takes no arguments.

Both **RemakeDisplay()** and **RethinkDisplay()** take several milliseconds to run and lock out all other tasks while they run. This can seriously degrade system performance, so do not use these routines lightly.

LIMITATIONS OF THE GRAPHICS SUBSYSTEM

If each of the visible screens does not have the same physical attributes, it may not be possible to display the data in its proper screen mode. *Screen coercion* is the technique that allows multiple screens with differing physical attributes to be displayed simultaneously. When a coerced screen is visible, its aspect ratio and colors may appear significantly changed. This is normal and the screen will be displayed correctly when it is the frontmost screen.

Hardware restrictions prevent certain types of displays. For instance, screens always use the full width of the display, regardless of the width of the overscan rectangle. This prevents any changes in display mode within a video line. Other modes, such as the VGA modes, require specific revisions of the custom chips and may not be available on all machines. See the "Graphics Primitives" chapter and the *Amiga Hardware Reference Manual* for more information on Amiga display organization and limitations.

Advanced Screen Programming

This section discusses how to perform double-buffering of Intuition screens, how to create a dual-playfield Intuition screen and other advanced topics.

DOUBLE BUFFERING

Double buffering of an Intuition screen involves the swapping of bitmaps of the screen, then updating the copper list to install the changes. The trick is that after installing the bitmaps to the screen the display is not updated to access these new bitmaps until the program explicitly updates the copper list. Any rendering performed before the copper list is updated will be rendered into the off-display bitmaps, appearing on the screen in completed form when the copper list is updated.

First, install the alternate bitmaps into the screen.

```
/* switch the bitmap so that we are drawing into the correct place */
screen->RastPort.BitMap          = myBitMaps[toggleFrame];
screen->ViewPort.RasInfo->BitMap = myBitMaps[toggleFrame];
```

Rendering may then take place into the off screen bitmaps by drawing into **screen->RastPort**.

The copper list of the screen is updated by calling **MakeScreen()**. This call refreshes the copper list, but does not install it into the system. Call **RethinkDisplay()** to install the new copper list so that the data is visible.

```
/* update the physical display to match the newly drawn bitmap. */
MakeScreen(screen); /* Tell intuition to do its stuff.         */
RethinkDisplay();   /* Intuition compatible MrgCop & LoadView   */
                    /*             it also does a WaitTOF().    */
```

Note that it is possible for the user to force the updating of the screen's copper list by dragging or depth-arranging the screen. This may cause information to be displayed before it is complete.

A complete example of double buffering a screen follows.

```
/* doublebuffer.c
** show the use of a double-buffered screen.
**
** SAS/C 5.10a
** lc -bl -cfist -v -y doublebuffer
** blink FROM LIB:c.o doublebuffer.o TO doublebuffer LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>


#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
/* characteristics of the screen */
#define SCR_WIDTH  (320)
#define SCR_HEIGHT (200)
#define SCR_DEPTH    (2)


/* Prototypes for our functions */
VOID   runDBuff(struct Screen *, struct BitMap ** );
struct BitMap **setupBitMaps( LONG, LONG, LONG );
VOID   freeBitMaps(struct BitMap **,LONG, LONG, LONG );
LONG   setupPlanes(struct BitMap *, LONG, LONG, LONG );
VOID    freePlanes(struct BitMap *, LONG, LONG, LONG );


struct Library *IntuitionBase = NULL;
struct Library *GfxBase       = NULL;

/*
** Main routine.  Setup for using the double buffered screen.
** Clean up all resources when done or on any error.
*/

VOID main(int argc, char **argv)
{
struct BitMap **myBitMaps;
struct Screen  *screen;
struct NewScreen myNewScreen;

IntuitionBase = OpenLibrary("intuition.library", 33L);
if ( IntuitionBase != NULL )
    {
    GfxBase = OpenLibrary("graphics.library", 33L);
    if ( GfxBase != NULL )
        {
        myBitMaps = setupBitMaps(SCR_DEPTH, SCR_WIDTH, SCR_HEIGHT);
        if ( myBitMaps != NULL )
            {
            /* Open a simple quiet screen that is using the first
            ** of the two bitmaps.
            */
            myNewScreen.LeftEdge=0;
            myNewScreen.TopEdge=0;
            myNewScreen.Width=SCR_WIDTH;
            myNewScreen.Height=SCR_HEIGHT;
            myNewScreen.Depth=SCR_DEPTH;
            myNewScreen.DetailPen=0;
            myNewScreen.BlockPen=1;
            myNewScreen.ViewModes=HIRES;
            myNewScreen.Type=CUSTOMSCREEN | CUSTOMBITMAP | SCREENQUIET;
            myNewScreen.Font=NULL;
            myNewScreen.DefaultTitle=NULL;
            myNewScreen.Gadgets=NULL;
            myNewScreen.CustomBitMap=myBitMaps[0];

            screen = OpenScreen(&myNewScreen);
            if (screen != NULL)
                {
                /* Indicate that the rastport is double buffered. */
                screen->RastPort.Flags = DBUFFER;

                runDBuff(screen, myBitMaps);

                CloseScreen(screen);
                }
            freeBitMaps(myBitMaps, SCR_DEPTH, SCR_WIDTH, SCR_HEIGHT);
            }
        CloseLibrary(GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

```
/*
** setupBitMaps(): allocate the bit maps for a double buffered screen.
*/
struct BitMap **setupBitMaps(LONG depth, LONG width, LONG height)
{
/* this must be static -- it cannot go away when the routine exits. */
static struct BitMap *myBitMaps[2];

myBitMaps[0] = (struct BitMap *) AllocMem((LONG)sizeof(struct BitMap), MEMF_CLEAR);
if (myBitMaps[0] != NULL)
    {
    myBitMaps[1] = (struct BitMap *)AllocMem((LONG)sizeof(struct BitMap), MEMF_CLEAR);
    if (myBitMaps[1] != NULL)
        {
        InitBitMap(myBitMaps[0], depth, width, height);
        InitBitMap(myBitMaps[1], depth, width, height);

        if (NULL != setupPlanes(myBitMaps[0], depth, width, height))
            {
            if (NULL != setupPlanes(myBitMaps[1], depth, width, height))
                return(myBitMaps);

            freePlanes(myBitMaps[0], depth, width, height);
            }
        FreeMem(myBitMaps[1], (LONG)sizeof(struct BitMap));
        }
    FreeMem(myBitMaps[0], (LONG)sizeof(struct BitMap));
    }
return(NULL);
}

/*
** runDBuff(): loop through a number of iterations of drawing into
** alternate frames of the double-buffered screen.  Note that the
** object is drawn in color 1.
*/
VOID runDBuff(struct Screen *screen, struct BitMap **myBitMaps)
{
WORD ktr, xpos, ypos;
WORD toggleFrame;

toggleFrame = 0;
SetAPen(&(screen->RastPort), 1);

for (ktr = 1; ktr < 200; ktr++)
    {
    /* Calculate a position to place the object, these
    ** calculations insure the object will stay on the screen
    ** given the range of ktr and the size of the object.
    */
    xpos = ktr;
    if ((ktr % 100) >= 50)
        ypos = 50 - (ktr % 50);
    else
        ypos = ktr % 50;

    /* switch the bitmap so that we are drawing into the correct place */
    screen->RastPort.BitMap          = myBitMaps[toggleFrame];
    screen->ViewPort.RasInfo->BitMap = myBitMaps[toggleFrame];

    /* Draw the objects.
    ** Here we clear the old frame and draw a simple filled rectangle.
    */
    SetRast(&(screen->RastPort), 0);
    RectFill(&(screen->RastPort), xpos, ypos, xpos+100, ypos+100);

    /* update the physical display to match the newly drawn bitmap. */
    MakeScreen(screen); /* Tell intuition to do its stuff.          */
    RethinkDisplay();   /* Intuition compatible MrgCop & LoadView   */
                        /*               it also does a WaitTOF().  */

    /* switch the frame number for next time through */
    toggleFrame ^= 1;
    }
}
```

```
/*
** freeBitMaps(): free up the memory allocated by setupBitMaps().
*/
VOID freeBitMaps(struct BitMap **myBitMaps, LONG depth, LONG width, LONG height)
{
freePlanes(myBitMaps[0], depth, width, height);
freePlanes(myBitMaps[1], depth, width, height);

FreeMem(myBitMaps[0], (LONG)sizeof(struct BitMap));
FreeMem(myBitMaps[1], (LONG)sizeof(struct BitMap));
}

/*
** setupPlanes(): allocate the bit planes for a screen bit map.
*/
LONG setupPlanes(struct BitMap *bitMap, LONG depth, LONG width, LONG height)
{
SHORT plane_num ;

for (plane_num = 0; plane_num < depth; plane_num++)
    {
    bitMap->Planes[plane_num] = (PLANEPTR)AllocRaster(width, height);
    if (bitMap->Planes[plane_num] != NULL )
        BltClear(bitMap->Planes[plane_num], (width / 8) * height, 1);
    else
        {
        freePlanes(bitMap, depth, width, height);
        return(NULL);
        }
    }
return(TRUE);
}

/*
** freePlanes(): free up the memory allocated by setupPlanes().
*/
VOID freePlanes(struct BitMap *bitMap, LONG depth, LONG width, LONG height)
{
SHORT plane_num ;

for (plane_num = 0; plane_num < depth; plane_num++)
    {
    if (bitMap->Planes[plane_num] != NULL)
        FreeRaster(bitMap->Planes[plane_num], width, height);
    }
}
```

DUAL-PLAYFIELD SCREEN EXAMPLE

This example shows how to create a dual-playfield display. Note that this technique is only valid for screen modes which support dual-playfield, do not try to convert other modes.

Setting up dual playfield mode in the **OpenScreen()** call is not the best method of obtaining a dual playfield viewport for a screen. It is better to open a standard screen, passing to Intuition (or letting Intuition create) only one of the playfield bitmaps (the front one). Next allocate and set up a second bitmap, its bitplanes, and a **RasInfo** structure installing these into the new screen's viewport. Update the viewport modes to include DUALPF and call **MakeScreen()** and **RethinkDisplay()**. This method, shown in the example below, keeps Intuition rendering (gadgets, menus, windows) in a single playfield.

```
/* dualplayfield.c
** Shows how to turn on dual-playfield mode in a screen.
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y dualplayfield
** blink FROM LIB:c.o dualplayfield.o TO dualplayfield LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY
```

```
#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <graphics/displayinfo.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>

VOID doDualPF ( struct Window * );
BOOL installDualPF( struct Screen *, struct RastInfo * );
VOID drawSomething( struct RastPort * );
VOID handleIDCMP ( struct Window * );
VOID removeDualPF( struct Screen *s );

struct Library *IntuitionBase;
struct Library *GfxBase;

VOID main(int argc, char **argv)
{
struct Window *win;
struct Screen *scr;

IntuitionBase = OpenLibrary("intuition.library",37);
if (IntuitionBase != NULL)
    {
    GfxBase = OpenLibrary("graphics.library", 37);
    if (GfxBase != NULL)
        {
        scr = OpenScreenTags(NULL,
                             SA_Depth,      2,
                             SA_DisplayID, HIRES_KEY,
                             SA_Title,      "Dual Playfield Test Screen",
                             TAG_END);
        if ( scr != NULL )
            {
            win = OpenWindowTags(NULL,
                                 WA_Title,        "Dual Playfield Mode",
                                 WA_IDCMP,        IDCMP_CLOSEWINDOW,
                                 WA_Width,        200,
                                 WA_Height,       100,
                                 WA_DragBar,      TRUE,
                                 WA_CloseGadget,  TRUE,
                                 WA_CustomScreen, scr,
                                 TAG_END);
            if ( win != NULL )
                {
                doDualPF(win);

                CloseWindow(win);
                }
            CloseScreen(scr);
            }
        CloseLibrary(GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
}


/*
** Allocate all of the stuff required to add dual playfield to a screen.
*/
VOID doDualPF(struct Window *win)
{
struct Screen   *myscreen;
struct RasInfo  *rinfo2;
struct BitMap   *bmap2;
struct RastPort *rport2;

myscreen = win->WScreen;   /* Find the window's screen */

/* Allocate the second playfield's rasinfo, bitmap, and bitplane */
rinfo2 = (struct RasInfo *) AllocMem(sizeof(struct RasInfo), MEMF_PUBLIC | MEMF_CLEAR);
if ( rinfo2 != NULL )
    {
```

```
    /* Get a rastport, and set it up for rendering into bmap2 */
    rport2 = (struct RastPort *) AllocMem(sizeof(struct RastPort), MEMF_PUBLIC );
    if (rport2 != NULL )
        {
        bmap2 = (struct BitMap *) AllocMem(sizeof(struct BitMap), MEMF_PUBLIC | MEMF_CLEAR);
        if (bmap2 != NULL )
            {
            InitBitMap(bmap2, 1, myscreen->Width, myscreen->Height);

            /* extra playfield will only use one bitplane here. */
            bmap2->Planes[0] = (PLANEPTR) AllocRaster(myscreen->Width, myscreen->Height);
            if (bmap2->Planes[0] != NULL )
                {
                InitRastPort(rport2);
                rport2->BitMap = rinfo2->BitMap = bmap2;

                SetRast(rport2, 0);

                if (installDualPF(myscreen,rinfo2))
                    {
                    /* Set foreground color; color 9 is color 1 for
                    ** second playfield of hi-res viewport
                    */
                    SetRGB4(&myscreen->ViewPort, 9, 0, 0xF, 0);

                    drawSomething(rport2);

                    handleIDCMP(win);

                    removeDualPF(myscreen);
                    }
                FreeRaster(bmap2->Planes[0], myscreen->Width, myscreen->Height);
                }
            FreeMem(bmap2, sizeof(struct BitMap));
            }
        FreeMem(rport2, sizeof(struct RastPort));
        }
    FreeMem(rinfo2, sizeof(struct RasInfo));
    }
}


/*
** Manhandle the viewport:
** install second playfield and change modes
*/
BOOL installDualPF(struct Screen *scrn, struct RastInfo *rinfo2)
{
ULONG screen_modeID;
BOOL return_code = FALSE;

screen_modeID = GetVPModeID(&(scrn->ViewPort));
if( screen_modeID != INVALID_ID )
    {
    /* you can only play with the bits in the Modes field
    ** if the upper half of the screen mode ID is zero!!!
    */
    if ( (screen_modeID & 0xFFFF0000L) == 0L )
        {
        return_code = TRUE;

        Forbid();

        /* Install rinfo for viewport's second playfield */
        scrn->ViewPort.RasInfo->Next = rinfo2;
        scrn->ViewPort.Modes |= DUALPF;

        Permit();

        /* Put viewport change into effect */
        MakeScreen(scrn);
        RethinkDisplay();
        }
    }
return(return_code);
}
```

```
/*
** Draw some lines in a rast port...This is used to get some data into
** the second playfield.  The windows on the screen will move underneath
** these graphics without disturbing them.
*/
VOID drawSomething(struct RastPort *rp)
{
int width, height;
int r, c;

width = rp->BitMap->BytesPerRow * 8;
height = rp->BitMap->Rows;

SetAPen(rp, 1);

for (r = 0; r < height; r += 40)
    {
    for (c = 0; c < width; c += 40)
        {
        Move(rp, 0L, r);
        Draw(rp, c, 0L);
        }
    }
}

/*
** simple event loop to wait for the user to hit the close gadget
** on the window.
*/
VOID handleIDCMP(struct Window *win)
{
BOOL done = FALSE;
struct IntuiMessage *message = NULL;
ULONG class;
ULONG signals;

while (!done)
    {
    signals = Wait(1L << win->UserPort->mp_SigBit);
    if (signals & (1L << win->UserPort->mp_SigBit))
        {
        while ((!done) &&
               (message = (struct IntuiMessage *)GetMsg(win->UserPort)))
            {
            class = message->Class;
            ReplyMsg((struct Message *)message);

            switch (class)
                {
                case IDCMP_CLOSEWINDOW:
                    done = TRUE;
                    break;
                }
            }
        }
    }
}

/*
** remove the effects of installDualPF().
** only call if installDualPF() succeeded.
*/
VOID removeDualPF(struct Screen *scrn)
{
Forbid();

scrn->ViewPort.RasInfo->Next = NULL;
scrn->ViewPort.Modes &= ~DUALPF;

Permit();

MakeScreen(scrn);
RethinkDisplay();
}
```

Other Screen Functions

Other screen functions provided by Intuition control screen depth arrangement, screen movement, the screen title bar and provide a visual ''error beep''.

SCREEN DEPTH ARRANGEMENT

ScreenToFront() and **ScreenToBack()** make a screen either the frontmost or the backmost screen. If an application needs to render into a screen before the screen becomes visible to the user, the screen may be opened behind all other screens and later moved to the front when ready with **ScreenToFront()**.

```
VOID ScreenToFront( struct Screen * )
VOID ScreenToBack ( struct Screen * )
```

Depth control of screens is also available through the depth arrangement gadget in the screen's title bar or through keyboard shortcuts. The N key with the Left-Amiga qualifier moves the Workbench screen to front. The M key with the Left-Amiga qualifier moves the frontmost screen to back. Repeated selection of Left-Amiga-M will cycle through available screens. These keys are processed through the keymap and will retain their value even if the key location changes.

SCREEN MOVEMENT AND SCROLLING

The **MoveScreen()** function moves the screen origin by the number of pixels specified in **dx** and **dy**.

```
VOID MoveScreen( struct Screen *myscreen, WORD dx, WORD dy )
```

Calls to **MoveScreen()** are asynchronous; the screen is not necessarily moved upon return of this function. If the calls happen too quickly, there may be unexpected results. One way to pace these calls is to call the function one time for each **IDCMP_INTUITICKS** event.

Screen movement is also available through the screen's drag gadget in the title bar and through a keyboard/mouse shortcut. Left-Amiga with the select button of the mouse anywhere within the screen will drag the screen (even if the title bar is totally concealed by a window). Dragging a screen down will reveal any screen(s) behind it. Screens are never revealed to the left, right or bottom of another screen.

Additionally, oversized screens may be moved with the new autoscroll feature of Release 2. With autoscroll, the screen is automatically scrolled as the pointer reaches one of the edges of the display. Autoscroll only works on the active screen.

Another screen movement feature added in Release 2 is screen *menu snap*. When a screen much larger than the viewing area is scrolled such that the upper left corner is not visible (scrolled down or to the right), menus may could be out of the visible portion of the screen. To prevent this, *menu snap* moves the screen to a position where the menus will be visible before rendering them. The screen appears to snap to the home position as the menus are selected, moving back when the operation is complete. If the Left-Amiga qualifier is held when the menus are selected then the screen will remain in the home position when the menu button is released.

The Intuition preferences editor, IControl, allows the user to change a number of Intuition features. Some of these features include the ability to globally disable menu snap, and to change the select qualifier for dragging the screen. See the User's Manual for more information on Preferences editors.

MISCELLANEOUS SCREEN FUNCTIONS

Three other functions used with screens are **DisplayBeep()**, **ShowTitle()** and **GetScreenData()**. **DisplayBeep()** flashes the screen colors to inform the user of an error or problem.

```
VOID DisplayBeep( struct Screen *myscreen )
```

Since not all users will have speakers attached to the system, **DisplayBeep()** can be used to provide a visible bell. **DisplayBeep()** can beep any single screen or, if **myscreen** is set to NULL, all screens.

ShowTitle() determines whether the screen's title bar will be displayed in front of or behind any backdrop windows on the screen.

```
VOID ShowTitle( struct Screen *myscreen, BOOL infront )
```

By default, the screen's title bar is set to display in front of backdrop windows. Call this function with **infront** set to FALSE to put the screen title bar behind backdrop windows. This can also be set when the screen is opened with the SA_ShowTitle tag.

Under 1.3 (V34) and earlier versions of the Amiga OS, applications used the **GetScreenData()** to get a copy of the Workbench **Screen** structure in order to examine its attributes.

```
success = BOOL GetScreenData( APTR buffer, UWORD bufsize, UWORD type, struct Screen *scr)
```

If successful, **GetScreenData()** copies a given **Screen** structure to a buffer supplied by the application. A copy of the Workbench **Screen** data can be obtained without knowing its location in memory using **GetScreenData(buf, sizeof(struct Screen), WBENCHSCREEN, NULL)**. However, for Release 2 and later versions of the operating system, this function may return some false information about the Workbench screen. This false screen information helps prevent older applications that used the call from malfunctioning when run in a Release 2 system that has Workbench set up with one of the newer modes.

Applications that want to get information on the Workbench screen should use **GetScreenData()** when run under 1.3 and **LockPubScreen()** when run under under Release 2. For more about **LockPubScreen()** and Workbench, see the section on ''Public Screen Functions'' earlier in this chapter.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of Intuition screens. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 3-10: Functions for Intuition Screens

Function	Description
OpenScreenTagList()	Open a screen.
OpenScreenTags()	Alternate calling sequence for OpenScreenTagList().
OpenScreen()	Pre-V36 open screen function.
CloseScreen()	Close an open screen.
MoveScreen()	Change the position of an open screen.
ScreenToBack()	Move a screen behind all other screens.
ScreenToFront()	Move a screen in front of all other screens.
ShowTitle()	Show the screen in front of through backdrop windows.
GetScreenDrawInfo()	Get the DrawInfo information for an open screen.
FreeScreenDrawInfo()	Free the DrawInfo information for a screen.
QueryOverscan()	Find overscan information for a specific display type.
LockPubScreen()	Obtain a lock on a public screen.
UnlockPubScreen()	Release a lock on a public screen.
NextPubScreen()	Return the name of the next public screen in the list.
PubScreenStatus()	Make a public screen private or private screen public.
LockPubScreenList()	Lock the public screen list (for a public screen utility).
UnlockPubScreenList()	Unlock the public screen list.
SetDefaultPubScreen()	Change the default public screen.
SetPubScreenModes()	Establish global public screen behavior.
GetDefaultPubScreen()	Copies the name of the default public screen to a buffer.
OpenWorkBench()	Open the Workbench screen, if closed.
CloseWorkBench()	Close the Workbench screen, if possible.
WBenchToBack()	Move the Workbench screen behind all other screens.
WBenchToFront()	Move the Workbench screen in front of all other screens.
GetScreenData()	Pre-V36 way to return information on an open screen.
ViewAddress()	Return the address of a screen's View.
ViewPortAddress()	Use **&screen->ViewPort** instead.
MakeScreen()	Low level screen handling--rebuild Copper list.
RethinkDisplay()	Low level screen handling--incorporate Copper list changes.
RemakeDisplay()	MakeScreen() for all screens, then RethinkDisplay().

Chapter 4
INTUITION WINDOWS

This chapter provides a general description of windows: how to open windows and define their characteristics; how to get the system gadgets for shaping, moving, closing, and depth arranging windows; how to handle window I/O; and how to preserve the display when windows get overlapped.

About Windows

Windows are rectangular display areas that open on screens. The window acts as a virtual terminal allowing a program to interact with the user as if it had the entire display all to itself.

Each window opens on a specific screen and takes certain characteristics, such as resolution, colors and display attributes, from that screen. These values cannot be adjusted on a window by window basis. Other window characteristics such as the text font are inherited from the screen but can be changed.

An application may open several windows at the same time on a single screen. The Workbench and other public (shareable) screens allow windows opened by different applications to coexist on the same screen.

Windows are moveable and can be positioned anywhere within the screen on which they exist. Windows may also have a title and borders containing various gadgets for controlling the window.

WINDOW SYSTEM GADGETS

Each window may have a number of system gadgets which allow the user to control window size, shape and arrangement. These gadgets are: the drag bar, the depth gadget, the sizing gadget, the zoom gadget and the close gadget.

The drag bar allows the user to change the position of the window with respect to the screen. The drag bar is in the top border of a window and occupies any space in the top border that is not used by other gadgets. The window may be dragged left, right, up and down on the screen, with the limitation that the entire window must remain within the screen's boundaries. This is done by positioning the pointer over the title bar, selecting the window and dragging to the new position. Window drag may be cancelled by pressing the right mouse button before the drag is completed.

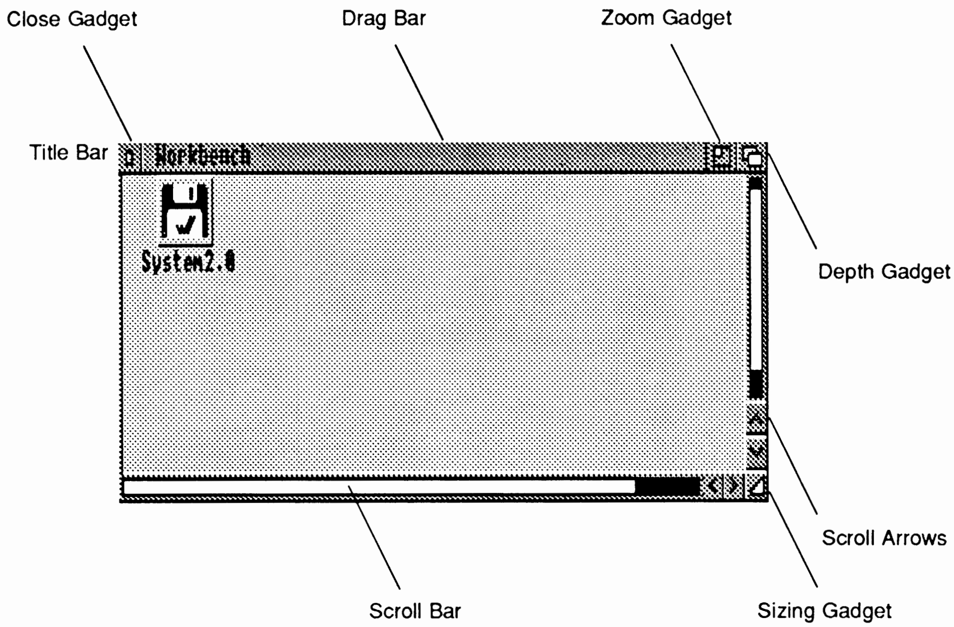


Figure 4-1: A Window with System Gadgets

The depth gadget allows the user to depth arrange a window with respect to other windows on the screen. The depth gadget is always positioned in the upper right corner of the window. Clicking the depth gadget will move the frontmost window behind all other windows. If the window is not the frontmost, it will be moved to the front. Selecting the depth gadget with the Shift qualifier always moves the window to the back (behind other windows).

The sizing gadget allows the user to change the size of the window. Sizing is subject to minimum and maximum values set by the application. Width and height are independent in a sizing operation. The sizing gadget is always positioned in the lower right corner of the window. It allows the user to drag this corner of the window to a new position relative to the upper left corner of the window, thus changing the width and height of the window. Window sizing using the sizing gadget may be cancelled by pressing the right mouse button before the size is completed.

The zoom gadget allows the user to quickly alternate between two preset window size and position values. The zoom gadget is always placed immediately to the left of the depth gadget. If there is no depth gadget on the window, the zoom gadget will still appear next to where the depth gadget would have been.

The close gadget performs no direct action on the window, rather it causes Intuition to send a message to the application to close the window. This allows the application to perform any required processing or to warn the user before it closes the window. The close gadget is always positioned in the upper left corner of the window.

THE ACTIVE WINDOW

There is only one window in the system active at any time. The active window receives all user input, including keyboard and mouse events. This is also known as the *input focus*, as all input is focused at this single point.

Some areas of the active window are displayed more boldly than those on inactive windows. The active window's borders are filled in with a color which is designed to stand out from the background while inactive windows have their borders filled with the background color. The specific coloring of active and inactive windows is dependent on the screen on which the window is opened. See the section ''DrawInfo and the 3D Look'' in the ''Intuition Screens'' chapter for more information.

Windows have two optional titles: one for the window and one for the screen. The window title appears in the top border of the window, regardless of whether the window is active or inactive. The window's screen title appears in the screen's title bar only when the window is active. This gives the user a secondary clue as to what application is active in the screen.

The active window's menus are displayed on the screen when the right mouse button (the menu button) is pressed. If the active window has no menus, then none will be displayed.

Each window may also have its own mouse-pointer image. Changing the active window will change the pointer to the one currently set for the new active window.

Basic Window Structures and Functions

This section introduces the basic data structures and functions an application uses to create an Intuition window. Intuition uses the **Window** data structure defined in *<intuition/intuition.h>* to represent windows. Most of Intuition's window functions use this structure in some way. Other related structures used to create and operate windows are summarized in Table 4-1.

Table 4-1: Data Structures Used with Intuition Windows

Structure Name	Description	Defined in Include File
Window	Main Intuition structure that defines a window	*<intuition/intuition.h>*
TagItem	General purpose parameter structure used to set up windows in V37	*<utility/tagitem.h>*
NewWindow	Parameter structure used to create a window in V34	*<intuition/intuition.h>*
ExtNewWindow	An extension to the **NewWindow** structure used in V37 for backward compatibility with older systems	*<intuition/intuition.h>*
Layer	A drawing rectangle that clips graphic operations falling within its boundaries	*<graphics/clip.h>*
RastPort	General purpose handle used for graphics library drawing operations.	*<graphics/rastport.h>*

Intuition's window system relies on the layers library and graphics library to implement many of its features. The **Window** structure is closely related to the **Layer** structure defined in *<graphics/clip.h>* and the **RastPort** structure defined in *<graphics/rastport.h>*. The system uses these structures to store drawing state data. In general, applications don't have to worry about the internal details of these structures but use them instead as convenient handles, passing them as arguments to lower-level functions. See the ''Layers Library'' and ''Graphics Primitives'' chapters for more information.

OPENING A WINDOW

A window is opened and displayed by a call to one of the **OpenWindow()** functions: **OpenWindow()**, **OpenWindowTagList()** or **OpenWindowTags()**.

```
struct Window *OpenWindowTagList( struct NewWindow *newWindow, struct TagItem *tagList );
struct Window *OpenWindowTags( struct NewWindow *newWindow, unsigned long tag1Type, ... );
struct Window *OpenWindow( struct NewWindow *newWindow );
```

The type of window and its attributes are specified in **NewWindow** or **TagItem** structures depending on which function is used. These functions all return a pointer to a new **Window** structure if they succeed. A NULL return indicates failure.

OpenWindowTagList() and **OpenWindowTags()** are available only in Release 2 (V36) and later versions of the OS. For these functions, window attributes are specified in **TagItem** structures which are paired data items specifying an attribute and its setting. (See the 'Utility Library' chapter for more information on **TagItems**.)

OpenWindow() is available in all versions of the OS. Window attributes can be specified using a **NewWindow** structure but only a limited set of window attributes are available this way. To support both the new window features of Release 2 and compatibility with older versions of the OS, use **OpenWindow()** with an extended version of the **NewWindow** structure named **ExtNewWindow**. See the WFLG_NW_EXTENDED flag description in the "Window Attributes" section below for more information on using **OpenWindow()** with the extended **NewWindow** structure.

Further references to **OpenWindow()** in this chapter will apply to all three functions. These calls are the only proper method for allocating a **Window** structure. The tag based versions are recommended for V36 and later versions of the OS. Use the **ExtNewWindow** structure with **OpenWindow()** to provide backward compatibility.

OpenWindowTagList() Example

Here's an example showing how to open a new window using the **OpenWindowTagList()** function with window attributes set up in a **TagItem** array.

```
;/* openwindowtags.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 openwindowtags.c
Blink FROM LIB:c.o,openwindowtags.o TO openwindowtags LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**
** openwindowtags.c - open a window using tags.
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <intuition/screens.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
#define MY_WIN_LEFT   (20)
#define MY_WIN_TOP    (10)
#define MY_WIN_WIDTH  (300)
#define MY_WIN_HEIGHT (110)

void handle_window_events(struct Window *);

struct Library *IntuitionBase;

struct TagItem win_tags[] =
    {
    {WA_Left,       MY_WIN_LEFT},
    {WA_Top,        MY_WIN_TOP},
    {WA_Width,      MY_WIN_WIDTH},
    {WA_Height,     MY_WIN_HEIGHT},
    {WA_CloseGadget,TRUE},
    {WA_IDCMP,      IDCMP_CLOSEWINDOW},
    {TAG_DONE, NULL},
    };

/*
** Open a simple window using OpenWindowTagList()
*/
VOID main(int argc, char **argv)
{
struct Window *win;

/* these calls are only valid if we have Intuition version 37 or greater */
IntuitionBase = OpenLibrary("intuition.library",37);
if (IntuitionBase!=NULL)
    {
    win = OpenWindowTagList(NULL,win_tags);
    if (win==NULL)
        {
        /* window failed to open */
        }
    else
        {
        /* window successfully opened here */
        handle_window_events(win);

        CloseWindow(win);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}

/* Normally this routine would contain an event loop like the one given
** in the chapter "Intuition Input and Output Methods".  Here we just
** wait for any messages we requested to appear at the Window's port.
*/
VOID handle_window_events(struct Window *win)
{
WaitPort(win->UserPort);
}
```

Setting Window Attributes

Depending on which function is used to open a window, the window's attributes may be specified using **TagItems**, or a **NewWindow** structure or an **ExtNewWindow** structure. In the code above, the window attributes are set up with an array of **TagItems**:

```
struct TagItem win_tags[] =
    {
    {WA_Left,       MY_WIN_LEFT},
    {WA_Top,        MY_WIN_TOP},
    {WA_Width,      MY_WIN_WIDTH},
    {WA_Height,     MY_WIN_HEIGHT},
    {WA_CloseGadget,TRUE},
    {WA_IDCMP,      IDCMP_CLOSEWINDOW},
    {TAG_DONE, NULL},
    };
```

These window attributes set the window's position (WA_Left, WA_Top) and size (WA_Width, WA_Height), request a close gadget on the window (WA_CloseGadget) and ask Intuition to send a message whenever the user activates the close gadget (WA_IDCMP).

Throughout this chapter window attributes are referred to by their **TagItem** ID name (the name is always prefixed with "WA_"). See the section below on "Window Attributes" for a complete list.

> *Old and New Flag Names.* The names used for IDCMP flags and window flags have been changed under Release 2. IDCMP flag names are now preceded by "IDCMP_". Likewise window flag names are now preceded by "WFLG_". The old names (and their new equivalents) are listed in *<intuition/iobsolete.h>*. You may want to refer to this file if you are working with example code written for V34 and older versions of the OS.

CLOSING WINDOWS

Call the **CloseWindow()** function to close a window, remove its imagery from the display, and clean up any system resources used by the window. Typically, you call **CloseWindow()** when Intuition informs you that the user has selected the window's close gadget but this is not a requirement nor does the window have to be active to be closed.

```
void CloseWindow( struct Window *window );
```

Pass this function a pointer to the **Window** structure returned by one of the **OpenWindow()** calls.

If you call **CloseWindow()** on the active window, the previously active window (if available) will become the active window. If the previously active window has already closed, then the window active prior to that window will become the active window. (Applications should not rely on this behavior. To make a specific window become active, call the **ActivateWindow()** function.)

Intuition does not automatically close a window when the user selects the close window gadget. Instead, Intuition sends your program a message about the user's action. The program can then perform whatever cleanup is necessary before closing the window with the **CloseWindow()** function.

WINDOWS AND SCREENS

Windows may be opened on one of three screen types: a custom screen, a public screen or the Workbench screen. A custom screen is one created and controlled by your application. Once you have set up a custom screen, you may open a window on it directly by calling one of the three open window functions.

To open a window on a custom screen, call **OpenWindowTagList()** (or **OpenWindowTags()**) with the WA_CustomScreen tag along with a pointer to the custom screen. This must be a pointer to a screen created by your application. For systems prior to Release 2, use the **OpenWindow()** call with **NewWindow.Type** set to CUSTOMSCREEN and **NewWindow.Screen** set to a pointer to your custom screen.

You may choose to open a window on an existing public (shareable) screen instead of setting up your own custom screen. Such windows are often referred to as *visitor* windows because they "visit" a screen managed by the system or another application.

For Workbench or other public screens that are not created and managed directly by your application, you must lock the screen before opening the window. This ensures that the screen remains open while your call to open the window is processed. One way to obtain a lock on a public screen is by calling the **LockPubScreen()** function (see the ''Intuition Screens'' chapter).

Use WA_PubScreenName with NULL to open a visitor window on the default public screen (normally the Workbench screen). If a name is provided and the named screen exists, the visitor window will open on that named screen. In this case the system locks the named screen for you so there is no need to call **LockPubScreen()** directly. The open window call will fail if it cannot obtain a lock on the screen. If the WA_PubScreenFallBack tag is TRUE, the window will open on the default public screen when WA_PubScreenName can't be found.

Another method to open a visitor window on a public screen is to use the WA_PubScreen tag along with a pointer to the **Screen** structure of the public screen obtained via **LockPubScreen()**.

The application may also request the name of the ''next'' public screen, which allows windows to ''jump'' between public screens. This is done by closing the application window on the first screen and opening a new window on the next screen. (See the ''Intuition Screens'' chapter for more information on public and custom screens.)

If no action is taken by the programmer to open the window on a specific screen, the window will open on the default public screen (normally the Workbench). This behavior is shown in the above example using **OpenWindowTagList()**.

There are two global modes which come into play when a visitor window is opened on a public screen. If the global mode SHANGHAI is set, Workbench application windows will be opened on the default public screen. A second global mode, POPPUBSCREEN, forces a public screen to be moved to the font when a visitor window opens on it. These modes can be changed using **SetPubScreenModes()**, however, these should only be set according to the preferences of the user.

Simple Window on a Public Screen Example

```
;/* winpubscreen.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 winpubscreen.c
Blink FROM LIB:c.o,winpubscreen.o TO winpubscreen LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**
** winpubscreen.c
** open a window on the default public screen (usually the Workbench screen)
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library *IntuitionBase;

/* our function prototypes */
VOID handle_window_events(struct Window *win);
```

```
/*
** Open a simple window on the default public screen,
** then leave it open until the user selects the close gadget.
*/
VOID main(int argc, char **argv)
{
struct Window *test_window = NULL;
struct Screen *test_screen = NULL;

IntuitionBase = OpenLibrary("intuition.library",37);
if (IntuitionBase)
    {
    /* get a lock on the default public screen */
    if (test_screen = LockPubScreen(NULL))
            {
            /* open the window on the public screen */
            test_window = OpenWindowTags(NULL,
                    WA_Left,  10,     WA_Top,     20,
                    WA_Width, 300,    WA_Height, 100,
                    WA_DragBar,         TRUE,
                    WA_CloseGadget,     TRUE,
                    WA_SmartRefresh,    TRUE,
                    WA_NoCareRefresh,   TRUE,
                    WA_IDCMP,           IDCMP_CLOSEWINDOW,
                    WA_Title,           "Window Title",
                    WA_PubScreen,       test_screen,
                    TAG_END);

            /* Unlock the screen.  The window now acts as a lock on
            ** the screen, and we do not need the screen after the
            ** window has been closed.
            */
            UnlockPubScreen(NULL, test_screen);

            /* if we have a valid window open, run the rest of the
            ** program, then clean up when done.
            */
            if (test_window)
                {
                handle_window_events(test_window);
                CloseWindow(test_window);
                }
            }
    CloseLibrary(IntuitionBase);
    }
}

/*
** Wait for the user to select the close gadget.
*/
VOID handle_window_events(struct Window *win)
{
struct IntuiMessage *msg;
BOOL done = FALSE;

while (! done)
    {
    /* We have no other ports of signals to wait on,
    ** so we'll just use WaitPort() instead of Wait()
    */
    WaitPort(win->UserPort);

    while ( (! done) &&
            (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        /* use a switch statement if looking for multiple event types */
        if (msg->Class == IDCMP_CLOSEWINDOW)
            done = TRUE;

        ReplyMsg((struct Message *)msg);
        }
    }
}
```

GRAPHICS AND TEXT IN WINDOWS

Applications can call functions in both the graphics library and the Intuition library to render images, lines, text and other graphic elements in windows. The graphics library provides primitive operations such as area fill, line drawing, text and animation.

The number of colors and the palette available in a window are defined by the screen in which the window opens. Applications should never change the palette of a screen unless the screen is a custom screen created by the application.

Graphics rendered into the window should respect the drawing pens defined for the screen. See the section on ''DrawInfo and the 3D Look'' in the ''Intuition Screens'' chapter for more information.

Default window fonts come from one of two places, depending on the screen on which the window opens. The window title font is always taken from the screen font. If the screen is opened with a font specified, either by specifying the tag **SA_Font** or the variable **NewScreen.Font**, then **Window.RPort->Font** is taken from the screen's font. Otherwise, the window's rastport's font is taken from **GfxBase->DefaultFont**. This information is available to the application if it opened the screen.

If the application did not open the screen, it has no way of knowing which font has been used for the window. Applications that require to know the window's font before the window is open must explicitly set the font (using **SetFont()**) for that window after opening it. In this case, the application may use any font it desires. It is recommended that applications use the screen's font if they support proportional fonts, and **GfxBase->DefaultFont** otherwise, as these fonts are generally the user's preference.

Intuition also provides a minimal high level interface to some of the functions in the Graphics library. This includes calls to draw lines, text and images. See the chapter entitled ''Intuition Images, Line Drawing and Text,'' for more information about using Intuition to render graphics.

WINDOW DIMENSIONS

The initial position and dimensions of the window are defined in the **OpenWindowTagList()** call. These values undergo error checking before the window is actually opened on the screen. If the dimensions are too big, the window will fail to open. (Or, you can use the WA_AutoAdjust tag if you want Intuition to move or size your window to fit.)

Maximum and minimum size values may also be defined, but are not required If the window does not have a sizing gadget. In setting these dimensions, bear in mind the horizontal and vertical resolutions of the screen in which the window will open.

The maximum dimensions of the window are unsigned values and may legally be set to the maximum by using the value 0xFFFF, better expressed as ''~0''. Using this value for the maximum dimensions allows the window to be sized to the full screen.

A Display Sized Window Example

A full screen window is not always desirable. If the user is working on a large, scrolling screen, they may only want a window the size of the visible display. The following example calculates the visible area on a screen and opens a window in that area. The example assumes that the screen display clip is as large or larger than text overscan (OSCAN_TEXT) which is set by the user. The window is opened in the text overscan area, not within the actual display clip that is used for the screen. Use **QueryOverscan()** to find the standard overscan rectangles (display clips) for a screen. Use the graphics library call **VideoControl()** to find the true display clip of the screen (see the chapter on "Graphics Primitives" for more information on **VideoControl()**). The **ViewPortExtra** structure contains the display clip information.

> *About Screen Coordinates.* The screen's actual position may not exactly equal the coordinates given in the **LeftEdge** and **TopEdge** fields of the **Screen** structure. This is due to hardware constraints that limit the fineness of the positioning of the underlying constructs. This may cause a window which is opened in the visible part of the screen to be incorrectly positioned by a small number of pixels in each direction. See the discussion of the screen's **LeftEdge** and **TopEdge** in the "Intuition Screens" chapter for more information.

```
;/* visiblewindow.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 visiblewindow.c
Blink FROM LIB:c.o,visiblewindow.o TO visiblewindow LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**
** open a window on the visible part of a screen, with the window as large
** as the visible part of the screen.  It is assumed that the visible part
** of the screen is OSCAN_TEXT, which how the user has set their preferences.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/displayinfo.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* Minimum window width and height:
** These values should really be calculated dynamically given the size
** of the font and the window borders.  Here, to keep the example simple
** they are hard-coded values.
*/
#define MIN_WINDOW_WIDTH  (100)
#define MIN_WINDOW_HEIGHT (50)

/* minimum and maximum calculations...Note that each argument is
** evaluated twice (don't use max(a++,foo(c))).
*/
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<=(b)?(a):(b))

struct Library *IntuitionBase;
struct Library *GfxBase;

/* our function prototypes */
VOID handle_window_events(struct Window *win);
VOID fullScreen(VOID);

/*
```

```
** open all the libraries and run the code.  Cleanup when done.
*/
VOID main(int argc, char **argv)
{
/* these calls are only valid if we have Intuition version 37 or greater */
if (GfxBase = OpenLibrary("graphics.library",37))
    {
    if (IntuitionBase = OpenLibrary("intuition.library",37))
        {
        fullScreen();

        CloseLibrary(IntuitionBase);
        }
    CloseLibrary(GfxBase);
    }
}


/*
** Open a window on the default public screen, then leave it open until the
** user selects the close gadget. The window is full-sized, positioned in the
** currently visible OSCAN_TEXT area.
*/
VOID fullScreen(VOID)
{
struct Window *test_window;
struct Screen *pub_screen;
struct Rectangle rect;
ULONG screen_modeID;
LONG width, height, left, top;

left  = 0;   /* set some reasonable defaults for left, top, width and height. */
top   = 0;   /* we'll pick up the real values with the call to QueryOverscan(). */
width = 640;
height= 200;

/* get a lock on the default public screen */
if (NULL != (pub_screen = LockPubScreen(NULL)))
    {
    /* this technique returns the text overscan rectangle of the screen that we
    ** are opening on.  If you really need the actual value set into the display
    ** clip of the screen, use the VideoControl() command of the graphics library
    ** to return a copy of the ViewPortExtra structure.  See the Graphics
    ** library chapter and Autodocs for more details.
    **
    ** GetVPModeID() is a graphics call...
    */

    screen_modeID = GetVPModeID(&pub_screen->ViewPort);
    if(screen_modeID != INVALID_ID)
        {
        if (QueryOverscan(screen_modeID, &rect, OSCAN_TEXT))
            {
            /* make sure window coordinates are positive or zero */
            left = max(0, -pub_screen->LeftEdge);
            top  = max(0, -pub_screen->TopEdge);

            /* get width and height from size of display clip */
            width  = rect.MaxX - rect.MinX + 1;
            height = rect.MaxY - rect.MinY + 1;

            /* adjust height for pulled-down screen (only show visible part) */
            if (pub_screen->TopEdge > 0)
                height -= pub_screen->TopEdge;

            /* insure that window fits on screen */
            height = min(height, pub_screen->Height);
            width  = min(width,  pub_screen->Width);

            /* make sure window is at least minimum size */
            width  = max(width,  MIN_WINDOW_WIDTH);
            height = max(height, MIN_WINDOW_HEIGHT);
            }
        }

    /* open the window on the public screen */
```

```
    test_window = OpenWindowTags(NULL,
            WA_Left, left,           WA_Width,  width,
            WA_Top,  top,            WA_Height, height,
            WA_CloseGadget, TRUE,
            WA_IDCMP,       IDCMP_CLOSEWINDOW,
            WA_PubScreen,   pub_screen,
            TAG_END);

    /* unlock the screen.  The window now acts as a lock on the screen,
    ** and we do not need the screen after the window has been closed.
    */
    UnlockPubScreen(NULL, pub_screen);

    /* if we have a valid window open, run the rest of the
    ** program, then clean up when done.
    */
    if (test_window)
        {
        handle_window_events(test_window);
        CloseWindow(test_window);
        }
    }
}


/*
** Wait for the user to select the close gadget.
*/
VOID handle_window_events(struct Window *win)
{
struct IntuiMessage *msg;
BOOL done = FALSE;

while (! done)
    {
    /* we only have one signal bit, so we do not have to check which
    ** bit(s) broke the Wait() (i.e. the return value of Wait)
    */
    Wait(1L << win->UserPort->mp_SigBit);

    while ( (! done) &&
            (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        /* use a switch statement if looking for multiple event types */
        if (msg->Class == IDCMP_CLOSEWINDOW)
            done = TRUE;

        ReplyMsg((struct Message *)msg);
        }
    }
}
```

WINDOW BORDER DIMENSIONS

Intuition automatically draws a border around a window unless directed otherwise, such as by setting the WFLG_BORDERLESS flag. Borderless windows may not have a window title or gadgets in the border (this includes the standard system gadgets). Otherwise they won't come out properly borderless.

The size of the border of an open window is available in the **Window** structure variables **BorderLeft**, **BorderTop**, **BorderRight** and **BorderBottom**. Intuition fills these in when the window is opened. To calculate the window border sizes *before* the window is opened you use information in the **Screen** structure as shown in the next listing.

> *Gadgets Can Change Border Sizes.* The following calculations do not take application border gadgets into account. If the program adds gadgets into the window's borders, Intuition will expand the borders to hold the gadgets.

```
if (NULL != (screen = LockPubScreen(NULL)))
    {
    top_border    = screen->WBorTop + screen->Font->ta_YSize + 1;
    left_border   = screen->WBorLeft;
    right_border  = screen->WBorRight;
    bottom_border = screen->WBorBottom;

    UnlockPubScreen(NULL,screen);
    }

/* if the sizing gadget is specified, then the border size must
** be adjusted for the border containing the gadget.  This may
** be the right border, the bottom border or both.
**
** We are using fixed values.  There is currently no system-approved
** method of finding this information before the window is opened.
** If you need to know these sizes BEFORE your window is opened,
** use the fixed values below.  Otherwise, use Window->BorderRight,
** etc. AFTER you have opened your window.
*/

/* values for non-lo-res screen */
right_border  = 18;  /* if sizing gadget in right border  */
bottom_border = 10;  /* if sizing gadget in bottom border */

/* values for lo-res screen */
right_border  = 13;  /* if sizing gadget in right border  */
bottom_border = 11;  /* if sizing gadget in bottom border */
```

Use the border sizes to position visual elements within the window. Coordinates may be offset into the window by the size of the top and left borders, for instance (x, y) becomes (x + **BorderLeft**, y + **BorderTop**). This may look clumsy, but it offers a way of avoiding a GimmeZeroZero window, which, although much more convenient to use, requires extra memory and degrades performance.

The right and bottom border values specify the width of these borders. The area within the borders of a window is defined as (**BorderLeft**, **BorderTop**) to (**Width** - 1 - **BorderRight**, **Height** - 1 - **BorderBottom**). The calculations subtract one from the height and width of the windows as positions count from zero, but dimensions count from one.

The window title bar is only available if one or more of the following is specified: window title, window drag gadget, window depth gadget, window close gadget or window zoom gadget. If none of these are specified, the top border will be much narrower.

Application gadgets may be added to the window border by setting a flag in the **Gadget** structure. A special flag must additionally be set to place gadgets into the borders of GimmeZeroZero windows. See the chapter "Intuition Gadgets," for more information about gadgets and their positioning. (Borderless windows have no visible border outlines and gadgets should not be placed in their borders.)

CHANGING WINDOW SIZE LIMITS

To change the sizing limits after the window has been opened, call **WindowLimits()** with the new values.

```
BOOL WindowLimits( struct Window *window, long widthMin, long heightMin,
                   unsigned long widthMax, unsigned long heightMax );
```

To maintain the current dimension, set the corresponding argument to 0. Out of range numbers are ignored. If the user is currently sizing the window, new limits take effect after the user releases the select button.

Communicating with Intuition

Intuition can notify an application when the user moves the mouse, makes a menu choice, selects an application gadget or changes the window's size. To find out about user activity from Intuition, there are two methods:

- Use the Intuition Direct Communications Message Port (IDCMP) system. Input events are received as standard Exec messages at a port Intuition creates for your window.
- Use the console.device to receive all input events as character sequences.

THE IDCMP

The IDCMP gives an application convenient access to many types of user input events through the Exec message and port system. Intuition input event messages include mouse and keyboard activity as well as high level events from menus and gadgets.

With the IDCMP, you specify the input events you want to know about when you open the window. The input events are specified with one or more of the IDCMP flags in *<intuition/intuition.h>*. Use the flags with the WA_IDCMP tag for the **OpenWindowTagList()** (or **OpenWindowTags()**) function. Or, set the flags in **NewWindow.IDCMPFlags** for the **OpenWindow()** function. If any IDCMP flags are set when the window is opened, Intuition automatically creates a message port for you to receive messages about user activity. If NULL is specified for IDCMP flags, no port is created. For more information on receiving messages from Intuition, see the IDCMP section in the chapter "Intuition Input and Output Methods."

THE CONSOLE DEVICE

An alternative to the message system used by the IDCMP is the console device. The console device gives your application input data translated to ASCII characters or ANSI escape sequences. Raw (untranslated) input is also available through the console device as ANSI escape sequences.

The console device also provides for convenient output of control codes and non-proportional (mono-spaced) text to the window. Output is character based, and includes capabilities such as automatic line wrapping and scrolling. The console device automatically formats and interprets the output stream. Output is kept within the window boundaries automatically so the application need not worry about overwriting the border (no GimmeZeroZero window required).

The console device must be opened by the application before it is used. See the chapter entitled "Intuition Input and Output Methods" or refer to the "Console Device" chapter of the *Amiga ROM Kernel Reference Manual: Devices* for more information about using the console device with your Intuition windows.

THE IDCMP AND THE ACTIVE WINDOW

On the Amiga, all input is directed to a single window called the *active* window. In general, changing the active window should be left up to the user. (The user activates a window by pressing the select button while the pointer is within the window boundaries.) If the active window is changed, the user may be confused if the change was not performed at their direction. Hence, new windows should be activated only when they open as a direct and synchronous response to the user's action. Existing windows should almost never be activated by the application.

An application can learn when one of its windows is activated or deactivated by setting the IDCMP flags IDCMP_ACTIVEWINDOW and IDCMP_INACTIVEWINDOW. When these flags are specified, the program will receive a message each time the user activates the window or causes the window to become inactive by activating some other window.

The application may specify that a window is to become active when it opens. This is done with the WA_Activate tag or by setting WFLG_ACTIVATE in **NewWindow.Flags** when the window is opened.

The application may also activate an existing window. This is done by calling the **ActivateWindow()** function, which will activate the window as soon as possible. Try to use this function only in response to user action since it may cause a shift in the input focus:

```
LONG ActivateWindow( struct Window *window );
```

This function call may have its action deferred. Do not assume that the selected window has become active when this call returns. Intuition will inform the application when this window has become active by sending an IDCMP_ACTIVEWINDOW message. Getting this message is the only supported way of tracking the activation status of your windows.

THE IDCMP AND GADGETS

One way for a user to communicate with a program running under Intuition is through the use of gadgets. There are two basic kinds of gadgets: system gadgets, which are predefined and managed by Intuition, and application gadgets.

System Gadgets

System gadgets on each window provide the user with the ability to manage the following aspects of the window: size, position and depth. These gadgets are managed by Intuition and the application does not need to take any action for them to operate properly. An additional system gadget is provided for the "close window" function. The close action is not directly managed by Intuition; selecting the close gadget will simply send a message to the application, which is responsible for closing the window.

All of these gadgets are optional, and independent of each other. The graphic representations of these gadgets are predefined, and Intuition always displays them in the same standard locations in the window borders.

The application may choose to be notified when the window changes size, or it may choose to control the timing of the sizing of the window. Controlling the timing of sizing operations is done through the use of the IDCMP_SIZEVERIFY message. IDCMP_SIZEVERIFY messages time out if the application does not respond fast enough. When these an IDCMP_SIZEVERIFY message times out the window sizing operation is cancelled by Intuition.

No information is available to the program on user changes to the depth arrangement of a window. However a refresh message will be sent if part of the window needs to be redrawn as a result of a change to the depth arrangement.

Notification of changes to the position of the window or the size of the window are available through the IDCMP_CHANGEWINDOW and IDCMP_NEWSIZE flags. The application specifies the initial size, the maximum and minimum limits for sizing, and whether the sizing gadget is contained in the right border,

bottom border or both borders. (See the section on "Border Dimensions" for information on how the specification of the sizing gadget affects the border sizes.)

The drag gadget has no imagery other than the implicit imagery of the title bar. Setting the window title does not interfere with drag gadget operation, nor does the drag gadget interfere with the display of the window title.

Application Gadgets

The application may place gadgets in windows to request various kinds of input from the user. These gadgets may be specified in the **OpenWindowTagList()** call, or they may be created and added to the window later. For details about creating and using gadgets, see the chapters on "Intuition Gadgets" and the "GadTools Library".

Window Types

There are three special window types: Backdrop, Borderless and GimmeZeroZero. Backdrop windows stay anchored to the back of the display. Borderless windows have no borders rendered by Intuition. GimmeZeroZero windows provide clipping to protect the borders from graphics rendered into the window.

These window types can be combined, although the combinations are not always useful. For instance, a borderless, backdrop window can be created; however, a borderless, GimmeZeroZero window does not make sense. A window is not required to be any of these types.

BACKDROP WINDOW TYPE

Backdrop windows open behind all other non-backdrop windows, but in front of other backdrop windows that might already be open. Depth arrangement of a backdrop window affects the order of the window relative to other backdrop windows, but backdrop windows always stay behind all non-backdrop windows. No amount of depth arrangement will ever move a non-backdrop window behind a backdrop window.

The only system gadget that can be attached to a backdrop window is the closewindow gadget. Application gadgets are not restricted in backdrop windows.

Backdrop windows may often be used in place of drawing directly into the display memory of a custom screen. Such a technique is preferred, as backdrop windows are compatible with the Intuition windowing system. Using a backdrop window eliminates the danger of writing to the screen memory at a "bad" time or at the wrong position and overwriting data in a window.

To provide a full screen display area that is compatible with the windowing system, create a full sized, borderless, backdrop window with no system gadgets. Use the **ShowTitle()** call to hide or reveal the screen's title bar, as appropriate. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a complete list of arguments for **ShowTitle()**.

Backdrop windows are created by specifying the WFLG_BACKDROP flag or the WA_Backdrop tag in the **OpenWindowTagList()** call.

BORDERLESS WINDOW TYPE

The borderless window type has no borders rendered by Intuition. Such a window will have no visual delineation from the rest of the display. Be aware that a Borderless window which does not cover the entire display may cause visual confusion for the user. When using a borderless window that does not cover the entire display, the application should provide some form of graphics to replace the borders provided by Intuition.

In general, none of the system gadgets or the window title should be specified for a borderless window, as they may cause at least part of the border to be rendered.

A typical application of a borderless window is to simulate graphics drawn directly into the screen, while remaining compatible with windows and menus. In this case, the application will often create a full sized, borderless, backdrop window.

Use the WFLG_BORDERLESS flag or the WA_Borderless tag to get this window type.

GIMMEZEROZERO WINDOW TYPE

GimmeZeroZero windows provide a window border layer separate from the main (inner) window layer. This allows the application to freely render into the window without worrying about the window border and its contents.

System gadgets and the window title are placed in the border layer. Application gadgets go into the inner window by default, but may be placed in the border. To position application gadgets in the border layer, the GTYP_GZZGADGET flag and the appropriate **Gadget** border flag must be set in the **Activation** field of the **Gadget**.

The top left coordinates of the inner window are always (0,0), regardless of the size or contents of the border, thus the name ''GimmeZeroZero.'' The application need not take the border size into account when rendering. The inner window always begins at (0,0) and extends to (**GZZWidth**,**GZZHeight**). The **GZZWidth** and **GZZHeight** variables are available in the **Window** structure.

The **GZZMouseX** and **GZZMouseY** variables provide the position of the mouse relative to the inner window. Note that the mouse positions in IDCMP_MOUSEMOVE events are always relative to the total window, even for GimmeZeroZero windows.

Requesters in a GimmeZeroZero window are also positioned relative to the inner window. See the chapter entitled ''Intuition Requesters and Alerts,'' for more information about requester location.

To specify a GimmeZeroZero window, set the WFLG_GIMMEZEROZERO flag or the WA_GimmeZeroZero tag in the **OpenWindowTagList()** call.

> *WARNING!* The GimmeZeroZero window uses more system resources than other window types because the window creates a separate layer for the border display. Using multiple GimmeZeroZero windows will quickly degrade performance in the positioning and sizing of windows.
>
> Applications should consider using *regions* as an alternative to GimmeZeroZero windows. See the ''Layers Library'' chapter, especially the **InstallClipRegion()** function, for information on setting up regions to limit graphics display in the window.

Preserving the Window Display

The layers library is what allows the display and manipulation of multiple overlapping rectangles, or *layers*. Intuition uses the layers library to manage its windows, by associating a layer to each window.

Each window is a virtual display. When rendering, the application does not have to worry about the current size or position of its window, and what other windows might be partly or fully obscuring its window. The window's **RastPort** is the handle to the its virtual display space. Intuition and graphics library rendering calls will recognize that this **RastPort** belongs to a layer, and act accordingly.

As windows are moved, resized, rearranged, opened, or closed, the on-screen representation changes. When part of a window which was visible now needs to appear in a new location, the layers library will move that imagery without involving the application. However, when part of a window that was previously obscured is revealed, or when a window is made larger, the imagery for the newly-visible part of the window needs to be redrawn. Intuition, through layers, offers three choices for how this is managed, trading off speed, memory usage, and application complexity.

- The most basic type of window is called *Simple Refresh.* When any graphics operation takes place in this kind of window, the visible parts are updated, but rendering to the obscured parts is discarded. When the window arrangement changes to reveal a previously obscured part of such a window, the application must refresh that area.

- Alternately, a window may be made *Smart Refresh*, which means that when rendering occurs, the system will not only update the visible parts of the window, but it will maintain the obscured parts as well, by using off-screen buffers. This means that when an obscured part of the window is revealed, the system will restore the imagery that belongs there. The application needs only to refresh parts of the window that appear when the window is made bigger. Smart Refresh windows use more memory than Simple Refresh windows (for the storage of obscured areas), but they are faster.

- The third kind of window is called *SuperBitMap.* In such a window, the system can refresh the window even when it is sized bigger. For this to work, the application must store a complete bitmap for the window's maximum size. Such a window is more work to manage, and uses yet more memory. SuperBitMap windows are used less often than the other two types.

Intuition helps your application manage window refresh. First, Intuition will take care of redrawing the window border and any system and application gadgets in the window. Your application never has to worry about that. Second, Intuition will notify your application when it needs to refresh its window (by sending the IDCMP_REFRESHWINDOW event). Third, Intuition provides functions that restrict your rendering to the newly-revealed (damaged) areas only, which speeds up your refresh rendering and makes it look cleaner.

The Intuition, layers, and graphics libraries work together to make rendering into and managing windows easy. You obtain your windows through Intuition, which uses the Layers library to manage the overlapping, resizing, and re-positioning of the window layers. The layers library is responsible for identifying the areas of each window that are visible, obscured but preserved off-screen, or obscured and not preserved. The rendering functions in the graphics library and Intuition library know how to render into the multiple areas that layers library establishes.

Note that you may not directly manipulate layers on an Intuition screen. You cannot create your own layers on an Intuition screen, nor can you use the layers movement, sizing, or arrangement functions on Intuition windows. Use the corresponding Intuition calls instead. Some other Layers library calls (such as the locking calls) are sometimes used on Intuition screens and windows.

DAMAGE REGIONS

The layers library and Intuition maintain a *damage region* for each window, which is the part of the window whose imagery is in need of repair, or refreshing. Several things can add areas of the window to the damage region:

- Revealing an obscured part of a Simple Refresh window adds that area to the damage region
- Sizing a Simple or Smart Refresh window bigger along either axis adds the new area to the damage region
- Resizing a Simple or Smart Refresh window (smaller or bigger) adds the old and new border areas, and the areas occupied by certain gadgets (those whose position or size depend on window size) to the damage region.

REFRESHING INTUITION WINDOWS

When the user or an application performs an Intuition operation which causes damage to a window, Intuition notifies that window's application. It does this by sending a message of the class IDCMP_REFRESHWINDOW to that window's IDCMP.

In response to this message, your application should update the damaged areas. Rendering proceeds faster and looks cleaner if it is restricted to the damaged areas only. The **BeginRefresh()/EndRefresh()** pair achieve that. The application should call **BeginRefresh()** for the window, and then do its rendering. Any rendering that would have gone into undamaged areas of the window is automatically discarded; only the area in need of repair is affected. Finally, the application should call **EndRefresh()**, which removes the restriction on rendering, and informs the system that the damage region has been dealt with. Even if your application intends to do no rendering, it must at least call **BeginRefresh()/EndRefresh()**, to inform the system that the damage region is no longer needed. If your application never needs to render in response to a refresh event, it can avoid having to call **BeginRefresh()/EndRefresh()** by setting the WFLG_NOCAREREFRESH flag or the WA_NoCareRefresh tag in the **OpenWindowTagList()** call.

Note that by the time that your application receives notification that refresh is needed, Intuition will have already refreshed your window's border and all gadgets in the window, as needed. Thus, it is unnecessary to use any of the gadget-refreshing functions in response to an IDCMP_REFRESHWINDOW event.

Operations performed between the **BeginRefresh()/EndRefresh()** pair should be restricted to simple rendering. All of the rendering functions in Intuition library and Graphics library are safe. Avoid **RefreshGList()** or **RefreshGadgets()**, or you risk deadlocking the computer. Avoid calls that may lock the LayerInfo or get complicated in Intuition, since **BeginRefresh()** leaves the window's layer or layers locked. Avoid **AutoRequest()** and **EasyRequest()**, and therefore all direct or indirect disk related DOS calls. See the "Intuition Gadgets" chapter for more information on gadget restrictions with **BeginRefresh()/EndRefresh()**.

Simple Refresh

For a Simple Refresh window, only those pixels actually on-screen are maintained by the system. When part of a Simple Refresh window is obscured, the imagery that was there is lost. As well, any rendering into obscured portions of such a window is discarded.

When part of the window is newly revealed (either because the window was just made larger, or because that part used to be obscured by another window), the application must refresh any rendering it wishes to appear into that part. The application will learn that refresh is needed because Intuition sends an IDCMP_REFRESHWINDOW event.

Smart Refresh

If a window is of the Smart Refresh type, then the system will not only preserve those pixels which are actually on-screen, but it will save all obscured pixels that are within the current window's size. The system will refresh those parts of the window revealed by changes in the overlapping with other windows on the screen, without involving the application. However, any part of the window revealed through the sizing of the window must be redrawn by the application. Again, Intuition will notify the application through the IDCMP_REFRESHWINDOW event.

Because the obscured areas are kept in off-screen buffers, Smart Refresh windows are refreshed faster than Simple Refresh windows are, and often without involving the application. Of course, for the same reason, they use more display memory.

SuperBitMap Refresh

The SuperBitMap refresh type allows the application to provide and maintain bitmap memory for graphics in the window. The bitmap can be any size as long as the window sizing limits respect the maximum size of the bitmap.

SuperBitMap windows have their own memory for maintaining all obscured parts of the window up to the size of the defined bitmap, including those parts outside of the current window. Intuition will update all parts of the window that are revealed through changes in sizing and changes in window overlapping. The application never needs to redraw portions of the window that were revealed by sizing or positioning windows in the screen.

SuperBitMap windows require the application to allocate a bitmap for use as off-screen memory, instead of using Intuition managed buffers. This bitmap must be as large as, or larger than, the inner window's maximum dimensions (that is, the window's outside dimensions less the border sizes).

SuperBitMap windows are almost always WFLG_GIMMEZEROZERO, which renders the borders and system gadgets in a separate bitmap. If the application wishes to create a SuperBitMap window that is not GimmeZeroZero, it must make the window borderless with no system gadgets, so that no border imagery is rendered by Intuition into the application's bitmap.

INTUITION REFRESH EVENTS

When using a Simple Refresh or a Smart Refresh windows, the program may receive refresh events, informing it to update the display. See the above discussion for information on when refresh events are sent.

A message of the class IDCMP_REFRESHWINDOW arrives at the IDCMP, informing the program of the need to update the display. The program must take some action when it receives a refresh event, even if it is just the acceptable minimum action described below.

On receiving a refresh event, **BeginRefresh()** must be called, then the program should redraw its display, and, finally, call **EndRefresh()**. The minimum required action is to call the **BeginRefresh()/EndRefresh()** pair. This allows Intuition and the Layers library keep things sorted and organized.

OPTIMIZED WINDOW REFRESHING

Bracketing the display updating in the **BeginRefresh()/EndRefresh()** pair automatically restricts all rendering to the "damaged" areas.

```
void BeginRefresh( struct Window *window );
void EndRefresh  ( struct Window *window, long complete );
```

These functions makes sure that refreshing is done in the most efficient way, only redrawing those portions of the window that really need to be redrawn. The rest of the rendering commands are discarded.

Operations performed between the **BeginRefresh()/EndRefresh()** pair should be restricted to simple rendering. All of the rendering functions in Intuition library and Graphics library are safe. Calls to **RefreshGadgets()** are not permitted. Avoid calls that may lock the **LayerInfo**, or get complicated in Intuition, since **BeginRefresh()** leaves the window's layer or layers locked. Avoid **AutoRequest()**, and therefore all direct or indirect disk related DOS calls. See the "Intuition Gadgets" chapter for more information on gadget restrictions with **BeginRefresh()/EndRefresh()**.

Certain applications do not need to receive refresh events, and can avoid having to call **BeginRefresh()** and **EndRefresh()** by setting the WFLG_NOCAREREFRESH flag or the WA_NoCareRefresh tag in the **OpenWindowTagList()** call.

The **EndRefresh()** function takes a boolean value as an argument (**complete** in the prototype above). This value determines whether refreshing is completely finished. When set to FALSE, further refreshing may be performed between subsequent **BeginRefresh()/ EndRefresh()** pairs. Set the boolean to TRUE for the last call to **EndRefresh()**.

It is critical that applications performing multiple **BeginRefresh()/EndRefresh()** pairs using **EndRefresh(win,FALSE)** hold layers locked through the entire process. The layer lock may only be released after the final call to **EndRefresh**(win,TRUE). See the "Layers Library" for more details.

The procedures outlined in this section take care of refreshing what is inside the window. Another function named **RefreshWindowFrame()** refreshes window borders, including the title region and gadgets:

```
void RefreshWindowFrame( struct Window *window );
```

Applications can use this function to update window borders after overwriting them with graphics.

SETTING UP A SUPERBITMAP WINDOW

SuperBitMap windows are created by setting the WFLG_SUPER_BITMAP flag, or by specifying the WA_SuperBitMap tag in the **OpenWindowTagList()** call. A pointer to an allocated and initialized **BitMap** structure must be provided.

A SuperBitMap window requires the application to allocate and initialize its own bitmap. This entails allocating a **BitMap** structure, initializing the structure and allocating memory for the bit planes.

Allocate a **BitMap** structure with the Exec **AllocMem()** function. Then use the graphics function **InitBitMap()** to initialize the **BitMap** structure:

```
void InitBitMap( struct BitMap *bitMap, long depth, long width, long height );
```

InitBitMap() fills in fields in the **BitMap** structure describing how a linear memory area is organized as a series of one or more rectangular bit-planes.

Once you have allocated and initialized the **BitMap** structure, use the graphics library function **AllocRaster()** to allocate the memory space for all the bit planes.

```
PLANEPTR AllocRaster( unsigned long width, unsigned long height );
```

The example listed in the next section shows how to allocate a **BitMap** structure, initialize it with **InitBitMap()** and use **AllocRaster()** function to set up memory for the bitplanes.

Graphics and Layers Functions for SuperBitMap Windows

The portion of the bitmap showing within a SuperBitMap window is controlled by the application. Initially, the window shows the bitmap starting from its origin (0,0) and clipped to fit within the window layer. The visible portion of the bitmap can be scrolled around within the window using the layers library **ScrollLayer()** function:

```
void ScrollLayer(LONG unused, struct Layer *layer, LONG dx, LONG dy)
```

Pass this function a pointer to the window's layer in **layer** and the scroll offsets in **dx** and **dy**. (A pointer to the window's layer can be obtained from **Window.RPort->Layer**.)

When rendering operations are performed in a SuperBitMap window, any rendering that falls outside window boundaries is done in the application's bitmap. Rendering that falls within window bounds is done in the screen's bitmap. Before performing an operation such as a save on the application bitmap, the graphics library function **SyncSBitMap()** should be called:

```
void SyncSBitMap(struct Layer *layer)
```

Pass this function a pointer to the window's layer. **SyncSBitMap()** copies the window contents to the corresponding part of the application bitmap, bringing it up to date. (If no rendering operations have been performed this call is not necessary.)

Similarly, after making any changes to the application bitmap such as loading a new one, the window's layer should be locked and the **CopySBitMap()** function should be called.

```
void CopySBitMap(struct Layer *)
```

This function copies the new information in the appropriate area of the underlying bitmap to the window's layer.

For more information about bitmaps and layers, see the "Graphics Primitives" and "Layers Library" chapters of this manual. Also see the *<graphics/clip.h>*, *<graphics/gfx.h>*, *<graphics/layers.h >*, graphics library and layers library sections of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

SuperBitMap Window Example

This example shows how to implement a superbitmap, and uses a host of Intuition facilities. Further reading of other Intuition and graphics chapters may be required for a complete understanding of this example.

```
;/* lines.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 lines.c
Blink FROM LIB:c.o,lines.o TO lines LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** lines.c -- implements a superbitmap with scroll gadgets
** This program requires V37, as it uses calls to OpenWindowTags(),
** LockPubScreen().
*/

/* Enforces use of new prefixed Intuition flag names */
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/layers_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

/* Random number function in amiga.lib (see amiga.lib.doc) */
UWORD RangeRand( unsigned long maxValue );

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

#define WIDTH_SUPER     (800)
#define HEIGHT_SUPER    (600)

#define UP_DOWN_GADGET    (0)
#define LEFT_RIGHT_GADGET (1)
#define NO_GADGET         (2)

#define MAXPROPVAL (0xFFFFL)

#define GADGETID(x) (((struct Gadget *)(msg->IAddress))->GadgetID)

#define LAYERXOFFSET(x) (x->RPort->Layer->Scroll_X)
#define LAYERYOFFSET(x) (x->RPort->Layer->Scroll_Y)

/* A string with this format will be found by the version command
** supplied by Commodore.  This will allow users to give version
** numbers with error reports.
*/
UBYTE vers[] = "$VER: lines 37.2";
```

```
struct Library *GfxBase;
struct Library *IntuitionBase;
struct Library *LayersBase;

struct Window      *Win = NULL;          /* window pointer */
struct PropInfo     BotGadInfo = {0};
struct Image        BotGadImage = {0};
struct Gadget       BotGad = {0};
struct PropInfo     SideGadInfo = {0};
struct Image        SideGadImage = {0};
struct Gadget       SideGad = {0};


/* Prototypes for our functions */
VOID initBorderProps(struct Screen *myscreen);
VOID doNewSize(void);
VOID doDrawStuff(void);
VOID doMsgLoop(void);
VOID superWindow(struct Screen *myscreen);


/*
** main
** Open all required libraries and get a pointer to the default public screen.
** Cleanup when done or on error.
*/
VOID main(int argc, char **argv)
{
struct Screen *myscreen;

/* open all of the required libraries for the program.
**
** require version 37 of the Intuition library.
*/

if (IntuitionBase = OpenLibrary("intuition.library",37L))
    {
    if (GfxBase = OpenLibrary("graphics.library",33L))
        {
        if (LayersBase = OpenLibrary("layers.library",33L))
            {
            /* LockPubScreen()/UnlockPubScreen is only available under V36
            ** and later... Use GetScreenData() under V34 systems to get a
            ** copy of the screen structure...
            */
            if (NULL != (myscreen = LockPubScreen(NULL)))
                {
                superWindow(myscreen);
                UnlockPubScreen(NULL,myscreen);
                }
            CloseLibrary(LayersBase);
            }
        CloseLibrary(GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
}


/*
** Create, initialize and process the super bitmap window.
** Cleanup if any error.
*/
VOID superWindow(struct Screen *myscreen)
{
struct BitMap *bigBitMap;
WORD planeNum;
WORD allocatedBitMaps;

/* set-up the border prop gadgets for the OpenWindow() call. */
initBorderProps(myscreen);

/* The code relies on the allocation of the BitMap structure with
** the MEMF_CLEAR flag.  This allows the assumption that all of the
** bitmap pointers are NULL, except those successfully allocated
** by the program.
```

```
*/
if (bigBitMap = AllocMem(sizeof(struct BitMap), MEMF_PUBLIC | MEMF_CLEAR))
    {
    InitBitMap(bigBitMap, myscreen->BitMap.Depth, WIDTH_SUPER, HEIGHT_SUPER);

    allocatedBitMaps = TRUE;
    for (planeNum = 0;
         (planeNum < myscreen->BitMap.Depth) && (allocatedBitMaps == TRUE);
             planeNum++)
        {
        bigBitMap->Planes[planeNum] = AllocRaster(WIDTH_SUPER, HEIGHT_SUPER);
        if (NULL == bigBitMap->Planes[planeNum])
            allocatedBitMaps = FALSE;
        }

    /* Only open the window if the bitplanes were successfully
    ** allocated.  Fail silently if they were not.
    */
    if (TRUE == allocatedBitMaps)
        {
        /* OpenWindowTags() and OpenWindowTagList() are only available
        ** when the library version is at least V36.  Under earlier
        ** versions of Intuition, use OpenWindow() with a NewWindow
        ** structure.
        */
        if (NULL != (Win = OpenWindowTags(NULL,
                WA_Width,  150,
                WA_Height, 4 * (myscreen->WBorTop + myscreen->Font->ta_YSize + 1),
                WA_MaxWidth,  WIDTH_SUPER,
                WA_MaxHeight, HEIGHT_SUPER,
                WA_IDCMP, IDCMP_GADGETUP | IDCMP_GADGETDOWN |
                    IDCMP_NEWSIZE | IDCMP_INTUITICKS | IDCMP_CLOSEWINDOW,
                WA_Flags, WFLG_SIZEGADGET | WFLG_SIZEBRIGHT | WFLG_SIZEBBOTTOM |
                    WFLG_DRAGBAR | WFLG_DEPTHGADGET | WFLG_CLOSEGADGET |
                    WFLG_SUPER_BITMAP | WFLG_GIMMEZEROZERO | WFLG_NOCAREREFRESH,
                WA_Gadgets, &(SideGad),
                WA_Title, &vers[6], /* take title from version string */
                WA_PubScreen, myscreen,
                WA_SuperBitMap, bigBitMap,
                TAG_DONE)))
            {
            /* set-up the window display */
            SetRast(Win->RPort,0); /* clear the bitplanes */
            SetDrMd(Win->RPort,JAM1);

          doNewSize();    /* adjust props to represent portion visible */
            doDrawStuff();

            /* process the window, return on IDCMP_CLOSEWINDOW */
            doMsgLoop();

            CloseWindow(Win);
            }
        }

    for (planeNum = 0; planeNum < myscreen->BitMap.Depth; planeNum++)
        {
        /* free only the bitplanes actually allocated... */
        if (NULL != bigBitMap->Planes[planeNum])
            FreeRaster(bigBitMap->Planes[planeNum], WIDTH_SUPER, HEIGHT_SUPER);
        }
    FreeMem(bigBitMap,sizeof(struct BitMap));
    }
}

/*
** Set-up the prop gadgets--initialize them to values that fit
** into the window border.  The height of the prop gadget on the side
** of the window takes the height of the title bar into account in its
** set-up. note the initialization assumes a fixed size "sizing" gadget.
**
** Note also, that the size of the sizing gadget is dependent on the
** screen resolution.  The numbers given here are only valid if the
** screen is NOT lo-res.  These values must be re-worked slightly
** for lo-res screens.
**
```

```
** The PROPNEWLOOK flag is ignored by 1.3.
*/
VOID initBorderProps(struct Screen *myscreen)
{
/* initializes the two prop gadgets.
**
** Note where the PROPNEWLOOK flag goes.  Adding this flag requires
** no extra storage, but tells the system that our program is
** expecting the new-look prop gadgets under 2.0.
*/
BotGadInfo.Flags     = AUTOKNOB | FREEHORIZ | PROPNEWLOOK;
BotGadInfo.HorizPot  = 0;
BotGadInfo.VertPot   = 0;
BotGadInfo.HorizBody = -1;
BotGadInfo.VertBody  = -1;

BotGad.LeftEdge     = 3;
BotGad.TopEdge      = -7;
BotGad.Width        = -23;
BotGad.Height       = 6;

BotGad.Flags        = GFLG_RELBOTTOM | GFLG_RELWIDTH;
BotGad.Activation   = GACT_RELVERIFY | GACT_IMMEDIATE | GACT_BOTTOMBORDER;
BotGad.GadgetType   = GTYP_PROPGADGET | GTYP_GZZGADGET;
BotGad.GadgetRender = (APTR)&(BotGadImage);
BotGad.SpecialInfo  = (APTR)&(BotGadInfo);
BotGad.GadgetID     = LEFT_RIGHT_GADGET;

SideGadInfo.Flags     = AUTOKNOB | FREEVERT | PROPNEWLOOK;
SideGadInfo.HorizPot  = 0;
SideGadInfo.VertPot   = 0;
SideGadInfo.HorizBody = -1;
SideGadInfo.VertBody  = -1;

/* NOTE the TopEdge adjustment for the border and the font for V36.
*/
SideGad.LeftEdge   = -14;
SideGad.TopEdge    = myscreen->WBorTop + myscreen->Font->ta_YSize + 2;
SideGad.Width      = 12;
SideGad.Height     = -SideGad.TopEdge - 11;

SideGad.Flags        = GFLG_RELRIGHT | GFLG_RELHEIGHT;
SideGad.Activation   = GACT_RELVERIFY | GACT_IMMEDIATE | GACT_RIGHTBORDER;
SideGad.GadgetType   = GTYP_PROPGADGET | GTYP_GZZGADGET;
SideGad.GadgetRender = (APTR)&(SideGadImage);
SideGad.SpecialInfo  = (APTR)&(SideGadInfo);
SideGad.GadgetID     = UP_DOWN_GADGET;
SideGad.NextGadget   = &(BotGad);
}


/*
** This function does all the work of drawing the lines
*/
VOID doDrawStuff()
{
WORD x1,y1,x2,y2;
WORD pen,ncolors,deltx,delty;

ncolors = 1 << Win->WScreen->BitMap.Depth;
deltx = RangeRand(6)+2;
delty = RangeRand(6)+2;

pen = RangeRand(ncolors-1) + 1;
SetAPen(Win->RPort,pen);
for(x1=0, y1=0, x2=WIDTH_SUPER-1, y2=HEIGHT_SUPER-1;
      x1 < WIDTH_SUPER;
            x1 += deltx, x2 -= deltx)
    {
    Move(Win->RPort,x1,y1);
    Draw(Win->RPort,x2,y2);
    }

pen = RangeRand(ncolors-1) + 1;
SetAPen(Win->RPort,pen);
for(x1=0, y1=0, x2=WIDTH_SUPER-1, y2=HEIGHT_SUPER-1;
```

```
      y1 < HEIGHT_SUPER;
            y1 += delty, y2 -= delty)
    {
    Move(Win->RPort,x1,y1);
    Draw(Win->RPort,x2,y2);
    }
}

/*
** This function provides a simple interface to ScrollLayer
*/
VOID slideBitMap(WORD Dx,WORD Dy)
{
ScrollLayer(0,Win->RPort->Layer,Dx,Dy);
}

/*
** Update the prop gadgets and bitmap positioning when the size changes.
*/
VOID doNewSize()
{
ULONG tmp;

tmp = LAYERXOFFSET(Win) + Win->GZZWidth;
if (tmp >= WIDTH_SUPER)
    slideBitMap(WIDTH_SUPER-tmp,0);

NewModifyProp(&(BotGad),Win,NULL,AUTOKNOB | FREEHORIZ,
    ((LAYERXOFFSET(Win) * MAXPROPVAL) /
        (WIDTH_SUPER - Win->GZZWidth)),
    NULL,
    ((Win->GZZWidth * MAXPROPVAL) / WIDTH_SUPER),
    MAXPROPVAL,
    1);

tmp = LAYERYOFFSET(Win) + Win->GZZHeight;
if (tmp >= HEIGHT_SUPER)
    slideBitMap(0,HEIGHT_SUPER-tmp);

NewModifyProp(&(SideGad),Win,NULL,AUTOKNOB | FREEVERT,
    NULL,
    ((LAYERYOFFSET(Win) * MAXPROPVAL) /
        (HEIGHT_SUPER - Win->GZZHeight)),
    MAXPROPVAL,
    ((Win->GZZHeight * MAXPROPVAL) / HEIGHT_SUPER),
    1);
}

/*
** Process the currently selected gadget.
** This is called from IDCMP_INTUITICKS and when the gadget is released
** IDCMP_GADGETUP.
*/
VOID checkGadget(UWORD gadgetID)
{
ULONG tmp;
WORD dX = 0;
WORD dY = 0;

switch (gadgetID)
    {
    case UP_DOWN_GADGET:
        tmp = HEIGHT_SUPER - Win->GZZHeight;
        tmp = tmp * SideGadInfo.VertPot;
        tmp = tmp / MAXPROPVAL;
        dY = tmp - LAYERYOFFSET(Win);
        break;
    case LEFT_RIGHT_GADGET:
        tmp = WIDTH_SUPER - Win->GZZWidth;
        tmp = tmp * BotGadInfo.HorizPot;
        tmp = tmp / MAXPROPVAL;
        dX = tmp - LAYERXOFFSET(Win);
        break;
    }
if (dX || dY)
    slideBitMap(dX,dY);
```

```
}

/*
** Main message loop for the window.
*/
VOID doMsgLoop()
{
struct IntuiMessage *msg;
WORD flag = TRUE;
UWORD currentGadget = NO_GADGET;

while (flag)
    {
    /* Whenever you want to wait on just one message port */
    /* you can use WaitPort(). WaitPort() doesn't require */
    /* the setting of a signal bit. The only argument it  */
    /* requires is the pointer to the window's UserPort   */
    WaitPort(Win->UserPort);
    while (msg = (struct IntuiMessage *)GetMsg(Win->UserPort))
        {
        switch (msg->Class)
            {
            case IDCMP_CLOSEWINDOW:
                flag = FALSE;
                break;
            case IDCMP_NEWSIZE:
                doNewSize();
            doDrawStuff();
                break;
            case IDCMP_GADGETDOWN:
                currentGadget = GADGETID(msg);
                break;
            case IDCMP_GADGETUP:
                checkGadget(currentGadget);
                currentGadget = NO_GADGET;
                break;
            case IDCMP_INTUITICKS:
                checkGadget(currentGadget);
                break;
            }
        ReplyMsg((struct Message *)msg);
        }
    }
}
```

The Window Structure

The **Window** structure is the main Intuition data structure used to represent a window. For the most part, applications treat this structure only as a handle. Window operations are performed by calling system functions that take **Window** as an argument instead of directly manipulating fields within the structure. However, there are some useful variables in a **Window** structure which are discussed in this section.

```
struct Window
    {
    struct Window *NextWindow;
    WORD LeftEdge, TopEdge, Width, Height;
    WORD MouseY, MouseX;
    WORD MinWidth, MinHeight;
    UWORD MaxWidth, MaxHeight;
    ULONG Flags;
    struct Menu *MenuStrip;
    UBYTE *Title;
    struct Requester *FirstRequest, *DMRequest;
    WORD ReqCount;
    struct Screen *WScreen;
    struct RastPort *RPort;
    BYTE BorderLeft, BorderTop, BorderRight, BorderBottom;
    struct RastPort *BorderRPort;
    struct Gadget *FirstGadget;
    struct Window *Parent, *Descendant;
```

```
UWORD *Pointer;
BYTE PtrHeight, PtrWidth;
BYTE XOffset, YOffset;
ULONG IDCMPFlags;
struct MsgPort *UserPort, *WindowPort;
struct IntuiMessage *MessageKey;
UBYTE DetailPen, BlockPen;
struct Image *CheckMark;
UBYTE *ScreenTitle;
WORD GZZMouseX, GZZMouseY, GZZWidth, GZZHeight;
UBYTE *ExtData;
BYTE *UserData;
struct Layer *WLayer;
struct TextFont *IFont;
ULONG MoreFlags;
};
```

LeftEdge, TopEdge, Width and Height

These variables reflect current position and size of the window. If the user sizes or positions the window, then these values will change. The position of the window is relative to the upper left corner of the screen.

MouseX, MouseY, GZZMouseX, GZZMouseY

The current position of the Intuition pointer with respect to the window, whether or not this window is currently the active one. For GimmeZeroZero windows, the GZZ variables reflect the position relative to the inner layer (see "Window Types" below). For normal windows, the GZZ variables reflect the position relative to the window origin after taking the borders into account.

ReqCount

Contains a count of the number of requesters currently displayed in the window. Do not rely on the value in this field, instead use IDCMP_REQSET and IDCMP_REQCLEAR to indirectly determine the number of open requesters in the window.

WScreen

A pointer to the **Screen** structure of the screen on which this window was opened.

RPort

A pointer to this window's **RastPort** structure. Use this **RastPort** pointer to render into your window with Intuition or graphics library rendering functions.

BorderLeft, BorderTop, BorderRight, BorderBottom

These variables describe the actual size of the window borders. The border size is not changed after the window is opened.

BorderRPort

With GimmeZeroZero windows, this variable points to the **RastPort** for the outer layer, in which the border gadgets are kept.

UserData

This pointer is available for application use. The program can attach a data block to this window by setting this variable to point to the data.

For a commented listing of the **Window** structure see *<intuition/intuition.h>* in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Window Attributes

This section discusses all window attributes. As mentioned earlier, a window's attributes may be specified with either **TagItems**, **NewWindow** or **ExtNewWindow** depending on how the window is opened.

Attributes are listed here by their **TagItem** ID name (**TagItem.ti_Tag**). For each tag item, the equivalent field setting in the **NewWindow** structure is also listed if it exists. Some window attributes specified with tags are available only in Release 2 and have no **NewWindow** equivalent.

EXTENDED NEW WINDOW

Of the three functions for opening a window, only **OpenWindow()** is present in all versions of the OS. This function takes a **NewWindow** structure as its sole argument. In order to allow applications to use the **OpenWindow()** call with Release 2 **TagItem** attributes, an extended version of the **NewWindow** structure has been created named **ExtNewWindow**.

Setting WFLG_NW_EXTENDED in the **NewWindow.Flags** field specifies to the **OpenWindow()** call that this **NewWindow** structure is really an **ExtNewWindow** structure. This is simply a standard **NewWindow** structure with a pointer to a tag list at the end. Since WFLG_NW_EXTENDED is ignored prior to V36, information provided in the tag list will be ignored by earlier versions of Intuition. Note that WFLG_NW_EXTENDED may not be specified in the WA_Flags tag.

WINDOW ATTRIBUTE TAGS

WA_Left, WA_Top, WA_Width and WA_Height

Describe where the window will first appear on the screen and how large it will be initially. These dimensions are relative to the top left corner of the screen, which has the coordinates (0,0).

WA_Left is the initial x position, or offset, from the left edge of the screen. The leftmost pixel is pixel 0, and values increase to the right. Equivalent to **NewWindow.LeftEdge**.

WA_Top is the initial y position, or offset, from the top edge of the screen. The topmost pixel is pixel 0, and values increase to the bottom. Equivalent to **NewWindow.TopEdge**.

WA_Width is the initial window width in pixels. Equivalent to **NewWindow.Width**.

WA_Height is the initial window height in lines. Equivalent to **NewWindow.Height**.

WA_DetailPen and WA_BlockPen

WA_DetailPen specifies the pen number for the rendering of window details like gadgets or text in the title bar. WA_BlockPen specifies the pen number for window block fills, like the title bar. These pens are also used for rendering menus. Equivalent to **NewWindow.DetailPen** and **NewWindow.BlockPen**.

The specific color associated with each pen number depends on the screen. Specifying -1 for these values sets the window's detail and block pen the same as the screen's detail and block pen.

Detail pen and block pen have largely been replaced starting with V36 by the pen array in the **DrawInfo** structure. See the section on "DrawInfo and the 3D Look" in the "Intuition Screens" chapter for more information.

WA_IDCMP

IDCMP flags tell Intuition what user input events the application wants to be notified about. The IDCMP flags are listed and described in the **OpenWindowTagList()** description in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* and in the chapter "Intuition Input and Output Methods" in this book. Equivalent to **NewWindow.IDCMPFlags.**

If any of these flags are set, Intuition creates a pair of message ports for the window (one internal to Intuition and one used by the application). These ports are for handling messages about user input events. If WA_IDCMP is NULL or unspecified, no IDCMP is created for this window.

The **ModifyIDCMP()** function can be used to change the window's IDCMP flags after it is open.

WA_Gadgets

A pointer to the first in the linked list of **Gadget** structures that are to be included in this window. These gadgets are application gadgets, not system gadgets. See the "Intuition Gadgets" chapter for more information. Equivalent to **NewWindow.FirstGadget**.

WA_Checkmark

A pointer to an **Image** structure, which is to be used as the checkmark image in this window's menus. To use the default checkmark, do not specify this tag or set this field to NULL. Equivalent to **NewWindow.CheckMark.**

WA_Title

A pointer to a NULL terminated text string, which is used as the window title and is displayed in the window's title bar.

Intuition draws the text using the colors defined in the **DrawInfo** pen array (**DrawInfo.dri_Pens**) and displays as much as possible of the window title, depending upon the current width of the title bar. Equivalent to **NewWindow.Title**. See the section on "DrawInfo and the 3D Look" in the "Intuition Screens" chapter for more information on the pen array.

The title is rendered in the screen's default font.

A title bar is added to the window if any of the properties WA_DragBar (WFLG_WINDOWDRAG), WA_DepthGadget (WFLG_WINDOWDEPTH), WA_CloseGadget (WFLG_WINDOWCLOSE) or WA_Zoom are specified, or if text is specified for a window title. If no text is provided for the title, but one or more of these system gadgets are specified, the title bar will be blank. Equivalent to **NewWindow.Title.**

WA_ScreenTitle

A pointer to a NULL terminated text string, which is used as the screen title and is displayed, when the window is active, in the screen's title bar. After the screen has been opened the screen's title may be changed by calling **SetWindowTitles()** (which is the only method of setting the window's screen title prior to V36).

WA_CustomScreen

A pointer to the **Screen** structure of a screen created by this application. The window will be opened on this screen. The custom screen must already be opened when the **OpenWindowTagList()** call is made. Equivalent to **NewWindow.Screen**, also implies **NewWindow.Type** of CUSTOMSCREEN.

WA_MinWidth, WA_MinHeight, WA_MaxWidth and WA_MaxHeight

These tags set the minimum and maximum values to which the user may size the window. If the flag WFLG_WINDOWSIZING is not set, then these variables are ignored. Values are measured in pixels. Use (~0) for the WA_MaxWidth (WA_MaxHeight) to allow for a window as wide (tall) as the screen. This is the complete screen, not the visible part or display clip.

Setting any of these variables to 0, will take the setting for that dimension from its initial value. For example, setting **MinWidth** to 0, will make the minimum width of this window equal to the initial width of the window.

Equivalent to **NewWindow.MinWidth**, **NewWindow.MinHeight**, **NewWindow.MaxWidth** and **NewWindow.MaxHeight**. Use the **WindowLimits()** function to change window size limits after the window is opened.

WA_InnerWidth and WA_InnerHeight

Specify the dimensions of the interior region of the window, i.e., inside the border, independent of the border widths. When using WA_InnerWidth and WA_InnerHeight an application will probably want to set WA_AutoAdjust (see below).

WA_PubScreen

Open the window as a visitor window on the public screen whose address is in the **ti_Data** field of the WA_PubScreen **TagItem**. To ensure that this screen remains open until **OpenWindowTagList()** has completed, the application must either be the screen's owner, have a window open on the screen, or use **LockPubScreen()**. Setting this tag implies screen type of PUBLICSCREEN.

WA_PubScreenName

Declares that the window is to be opened as a visitor on the public screen whose name is pointed to by the **ti_Data** field of the WA_PubScreenName **TagItem**. The **OpenWindowTagList()** call will fail if it cannot obtain a lock on the named public screen and no fall back name (WA_PubScreenFallBack) is specified. Setting this tag implies screen type of PUBLICSCREEN.

WA_PubScreenFallBack

A Boolean, specifies whether a visitor window should "fall back" to the default public screen (or Workbench) if the named public screen isn't available This tag is only meaningful when used in conjunction with WA_PubScreenName.

WA_Zoom

Pointer to an array of four WORDs, the initial LeftEdge, TopEdge, Width and Height values for the alternate zoom position and size. It also specifies that the application wants a zoom gadget for the window, whether or not it has a sizing gadget.

A zoom gadget is always supplied to a window if it has both depth and sizing gadgets. This tag allows the application to open a window with a zoom gadget when the window does not have both the depth and sizing gadgets.

WA_MouseQueue

An initial value for the mouse message backlog limit for this window. The **SetMouseQueue()** function will change this limit after the window is opened.

WA_RptQueue

An initial value of repeat key backlog limit for this window.

BOOLEAN WINDOW ATTRIBUTE TAGS

These boolean window tags are alternatives to the **NewWindow.Flags** bit fields with similar names. Unlike the tags discussed above, the **ti_Data** field of these **TagItems** is set to either TRUE or FALSE.

WA_SizeGadget

Specifying this flag tells Intuition to add a sizing gadget to the window. Intuition places the sizing gadget in the lower right corner of the window. By default, the right border is adjusted to accommodate the sizing gadget, but the application can specify one of the following two flags to change this behavior. The WFLG_SIZEBRIGHT flag puts the sizing gadget in the right border. The WFLG_SIZEBBOTTOM flag puts the sizing gadget in the bottom border. Both flags may be specified, placing the gadget in both borders. Equivalent to **NewWindow.Flags** WFLG_SIZEGADGET.

WA_SizeBRight

Place the size gadget in the right border. Equivalent to **NewWindow.Flags** WFLG_SIZEBRIGHT.

WA_SizeBBottom

Place the size gadget in the bottom border. Equivalent to **NewWindow.Flags** WFLG_SIZEBBOTTOM.

WA_DragBar

This flag turns the entire title bar of the window into a drag gadget, allowing the user to position the window by clicking in the title bar and dragging the mouse. Equivalent to **NewWindow.Flags** WFLG_DRAGBAR.

WA_DepthGadget

Setting this flag adds a depth gadget to the window. This allows the user to change the window's depth arrangement with respect to other windows on the screen. Intuition places the depth gadget in the upper right corner of the window. Equivalent to **NewWindow.Flags** WFLG_DEPTHGADGET.

WA_CloseGadget

Setting this flag attaches a close gadget to the window. When the user selects this gadget, Intuition transmits a message to the application. It is up to the application to close the window with a **CloseWindow()** call. Intuition places the close gadget in the upper left corner of the window. Equivalent to **NewWindow.Flags** WFLG_CLOSEGADGET.

WA_ReportMouse

Send mouse movement events to the window as x,y coordinates. Also see the description of the IDCMP flag IDCMP_MOUSEMOVE, in the chapter "Intuition Input and Output Methods." Equivalent to **NewWindow.Flags** WFLG_REPORTMOUSE.

The WFLG_REPORTMOUSE flag in the **Flags** field of the **Window** structure may be modified on the fly by the program. Changing this flag must be done as an atomic operation. Most compilers generate atomic code for operations such as `window->flags |= WFLG_REPORTMOUSE` or `window->flags &= ~WFLG_REPORTMOUSE`. If you are unsure of getting an atomic operation from your compiler, you may wish to do this operation in assembler, or bracket the code with a **Forbid()/Permit()** pair.

The use of the **ReportMouse()** function is strongly discouraged, due to historic confusion over the parameter ordering.

WA_NoCareRefresh

This window does not want IDCMP_REFRESHWINDOW events. Set this flag to prevent the window from receiving refresh window messages. Equivalent to **NewWindow.Flags** WFLG_NOCAREREFRESH. Intuition will manage **BeginRefresh()** and **EndRefresh()** internally.

WA_Borderless

Open a window with no borders rendered by Intuition. Equivalent to **NewWindow.Flags** WFLG_BORDERLESS.

Use caution setting this flag, as it may cause visual confusion on the screen. Also, some borders may be rendered if any of the system gadgets are requested, if text is supplied for the window's title bar, or if any of application gadgets are in the borders.

WA_Backdrop

Make this window a Backdrop window. Equivalent to **NewWindow.Flags** WFLG_BACKDROP.

WA_GimmeZeroZero

Set this tag to create a GimmeZeroZero window. GimmeZeroZero windows have the window border and border gadgets rendered into an extra layer. This extra layer slows down window operations, thus it is recommended that applications only use GimmeZeroZero windows when they are required. For clipping graphics to the area within the borders of a window, see the discussion of "Regions" in the "Layers Library" chapter. Equivalent to **NewWindow.Flags** WFLG_GIMMEZEROZERO.

WA_Activate

Activate the window when it opens. Equivalent to **NewWindow.Flags** WFLG_ACTIVATE. Use this flag carefully, as it can change where the user's input is going.

WA_RMBTrap

Catch right mouse button events for application use. Set this flag to disable menu operations for the window. When set, right mouse button events will be received as IDCMP_MOUSEBUTTONS with the MENUUP and MENUDOWN qualifiers. Equivalent to **NewWindow.Flags** WFLG_RMBTRAP.

The WFLG_RMBTRAP flag in the **Window** structure **Flags** field may be modified on the fly by the program. Changing this flag must be done as an atomic operation, as Intuition can preempt a multistep set or clear operation. An atomic operation can be done in assembler, using 68000 instructions that operate directly on memory. If you are unsure of generating such an instruction, place the operation within a **Forbid()/Permit()** pair. This will ensure proper operation by disabling multitasking while the flag is being changed.

WA_SimpleRefresh

The application program takes complete responsibility for updating the window. Only specify if TRUE. Equivalent to **NewWindow.Flags** WFLG_SIMPLE_REFRESH.

WA_SmartRefresh

Intuition handles all window updating, except for parts of the window revealed when the window is sized larger. Only specify if TRUE. Equivalent to **NewWindow.Flags** WFLG_SMART_REFRESH.

WA_SmartRefresh windows without a sizing gadget will never receive refresh events due to the user sizing the window. However, if the application sizes the window through a call like **ChangeWindowBox()**, **ZipWindow()** or **SizeWindow()**, a refresh event may be generated. Use WA_NoCareRefresh to disable refresh events.

WA_SuperBitMap

This is a pointer to a **BitMap** structure for a SuperBitMap window. The application will be allocating and maintaining its own bitmap. Equivalent to **NewWindow.BitMap**. Setting this tag implies the WFLG_SUPER_BITMAP property.

For complete information about SuperBitMap, see "Setting Up a SuperBitMap Window" in this chapter.

WA_AutoAdjust

Allow Intuition to change the window's position and dimensions in order to fit it on screen. The window's position is adjusted first, then the size. This property may be especially important when using WA_InnerWidth and WA_InnerHeight as border size depends on a user specified font.

WA_MenuHelp (new for V37, ignored by V36)

Enables IDCMP_MENUHELP: pressing Help during menus will return IDCMP_MENUHELP message. See the "Intuition Menus" chapter for more information.

WA_Flags

Multiple initialization of window flags, equivalent to **NewWindow.Flags**. Use the WFLG_ constants to initialize this field, multiple bits may be set by ORing the values together.

WA_BackFill

Allows you to specify a backfill hook for your window's layer. See the description of **CreateUpFrontHookLayer()** in the "Layers Library" chapter. Note that this tag is implemented in V37, contrary to what some versions of the include files may say.

Other Window Functions

This section contains a brief overview of other Intuition functions that affect windows. For a complete description of all Intuition functions, see the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

MENUS AND THE ACTIVE WINDOW

Menus for the active window will be displayed when the user presses the menu button on the mouse. Menus may be disabled for the window by not providing a menu strip, or by clearing the menus with **ClearMenuStrip()**. Similarly, if the active window has WFLG_RMBTRAP set, the menu button will not bring up the menus.

Two other functions, **SetMenuStrip()** and **ResetMenuStrip()**, are used to attach or update the menu strip for a window.

```
void ClearMenuStrip( struct Window *window );
BOOL SetMenuStrip( struct Window *window, struct Menu *menu );
BOOL ResetMenuStrip( struct Window *window, struct Menu *menu );
```

If **SetMenuStrip()** has been called for a window, **ClearMenuStrip()** must be called before closing the window. After **ClearMenuStrip()** has been called, the user can no longer access menus for this window. See the chapter "Intuition Menus," for complete information about setting up menus.

REQUESTERS IN THE WINDOW

Requesters are temporary sub-windows, usually containing several gadgets, used to confirm actions, access files, or adjust the options of a command the user has just given. **Request()** creates and activates a requester in the window. **EndRequest()** removes the requester from the window.

```
BOOL Request( struct Requester *requester, struct Window *window );
void EndRequest( struct Requester *requester, struct Window *window );
```

For simple requesters in a format that matches system requesters, two new functions have been added to Release 2:

```
LONG EasyRequestArgs( struct Window *window, struct EasyStruct *easyStruct,
                      ULONG *idcmpPtr, APTR args );
LONG EasyRequest( struct Window *window, struct EasyStruct *easyStruct,
                      ULONG *idcmpPtr, APTR arg1, ... );
```

The **EasyRequest()** functions support requesters with one or more gadgets automatically providing a layout that is sensitive to the current font and screen resolution. See the chapter ''Intuition Requesters and Alerts'' for more information on using requester functions.

PROGRAM CONTROL OF WINDOW ARRANGEMENT

MoveWindow(), **SizeWindow()**, **WindowToFront()** and **WindowToBack()** allow the program to modify the size and placement of its windows. These calls are available in all versions of the operating system.

MoveWindowInFrontOf(), **ChangeWindowBox()** and **ZipWindow()** have been added in Release 2 to provide more flexible control over the size and placement of windows.

All of these functions are asynchronous. The window will not be affected by them immediately, rather, Intuition will act on the request the next time it receives an input event. Currently this happens at a minimum rate of ten times per second, and a maximum of sixty times per second. There is no guarantee that the operation has taken place when the function returns. In some cases, there are IDCMP messages which will inform the application when the change has completed (for example, an IDCMP_NEWSIZE event indicates that a resize operation has completed).

Use the **MoveWindow()** function to move a window to a new position in the screen. Use **SizeWindow()** to change the size of the window:

```
void MoveWindow( struct Window *window, long dx, long dy );
void SizeWindow( struct Window *window, long dx, long dy );
```

Note that both **MoveWindow()** and **SizeWIndow()** take the amount of change in each axis (delta values instead of absolute coordinates). To specify the coordinates as absolute numbers, use **ChangeWindowBox()**. The **SizeWindow()** function will respect the window's maximum and minimum dimensions only if the window has a sizing gadget.

A new function in Release 2, **ChangeWindowBox()**, allows an application to change the window size and position in a single call:

```
void ChangeWindowBox( struct Window *window, long left, long top, long width, long height );
```

Note that the position and size values are absolutes and not deltas. The window's maximum and minimum dimensions are always respected.

To depth arrange windows under program control, use **WindowToFront()** and **WindowToBack()**:

```
void WindowToFront( struct Window *window );
void WindowToBack( struct Window *window );
```

WindowToFront() depth arranges a given window in front of all other windows on its screen. **WindowToBack()** depth arranges a given window behind all other windows on its screen.

To move a window in front of a specific, given window (as opposed to all windows), use **MoveWindowInFrontOf()**:

```
void MoveWindowInFrontOf( struct Window *window, struct Window *behindWindow );
```

MoveWindowInFrontOf() is a new call provided in Release 2 and is not available in older versions of the OS.

To toggle the window size between its two zoom settings use **ZipWindow()**. This performs the same action that occurs when the user selects the zoom gadget:

```
void ZipWindow( struct Window *window );
```

The two zoom settings are the initial size and position of the window when it was first opened and the alternate position specified with the WA_Zoom tag. If no WA_Zoom tag is provided, the alternate position is taken from the window's minimum dimensions, unless the window was opened at its minimum dimension. In that case, the alternate position is taken from the window's maximum dimension. **ZipWindow()** is a new call provided in Release 2 and is not available in older versions of the OS.

CHANGING THE WINDOW OR SCREEN TITLE

Each window has its own window title and local screen title. The window title, if specified, is always displayed in the window. The local screen title, if specified, is only displayed in the screen's title bar when the window is active. If the window does not specify a local screen title, then the default screen title is used in the screen title bar when this window is active.

```
void SetWindowTitles( struct Window *window, UBYTE *windowTitle, UBYTE *screenTitle );
```

This function changes the window title or local screen title for the given window. Both **windowTitle** and **screenTitle** can be set to -1, NULL or a NULL terminated string. Specifying -1 will not change the title from the current value. Specifying NULL will clear the window title or reset the screen title to the default title for the screen.

CHANGING MESSAGE QUEUE LIMITS

Starting with V36, windows have limits on the number of mouse movement and repeat key messages that may be waiting at their IDCMP at any time. These queue limits prevent the accumulation of these messages, which may arrive at the IDCMP message port in large numbers.

Once a queue limit is reached, further messages of that type will be discarded by Intuition. The application will never hear about the discarded messages; they are gone forever. (Note that only mouse move and key repeat messages are limited this way. Other types of messages will still be added to the port.) Messages of the limited type will arrive at the port again after the application has replied to one of the messages in the queue.

The queue limits are independent of each other. Having reached the limit for one type of message does not prevent other types of messages (that have not yet reached their queuing limits) from being added to the IDCMP. Note that the queues apply only to the IDCMP and not to messages received directly via an input handler or from the console device.

Order of event arrival is not a factor in the message count. Messages may be sequential or interspersed with other events--only the number of messages of the specific type waiting at the IDCMP matters.

The WA_RptQueue tag allows setting an initial value for the repeat key backlog limit for the window. There is no function to change this value as of V37. The default value for WA_RptQueue is 3.

The WA_MouseQueue tag allows setting an initial value for the mouse message backlog limit for the window. The default value for WA_MouseQueue is 5. The number may later be changed with a call to **SetMouseQueue()**:

```
LONG SetMouseQueue( struct Window *window, unsigned long queueLength );
```

Note that real information may be lost if the queue fills and Intuition is forced to discard messages. See the chapter "Intuition Mouse and Keyboard" for more information.

CHANGING POINTER POSITION REPORTS

Pointer position messages to a window may be turned on and off by simply setting or clearing the WFLG_REPORTMOUSE flag bit in **Window->Flags**, in an atomic way, as explained for the WA_RMBTrap tag in the "Window Attributes" section above. Using this direct method of setting the flag avoids the historic confusion on the ordering of the arguments of the **ReportMouse()** function call.

Mouse reporting may be turned on even if mouse movements were not activated when the window was opened. The proper IDCMP flags must be set for the window to receive the messages. See the chapter "Intuition Mouse and Keyboard" for more details on enabling mouse reporting in an application.

CUSTOM POINTERS

The active window also has control over the pointer. If the active window changes the image for the pointer using the functions **SetPointer()** or **ClearPointer()**, the pointer image will change:

```
void SetPointer( struct Window *window, UWORD *pointer, long height,
                 long width, long xOffset, long yOffset );

void ClearPointer( struct Window *window );
```

SetPointer() sets up the window with a sprite definition for a custom pointer. If the window is active, the change takes place immediately. The pointer will not change if an inactive window calls **SetPointer()**. In this way, each window may have its own custom pointer that is displayed only when the window is active.

ClearPointer() clears the custom pointer from the window and restores it to the default Intuition pointer, which is set by the user. Setting a pointer for a window is discussed further in the chapter "Intuition Mouse and Keyboard".

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of Intuition windows. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 4-2: Functions for Intuition Windows

Function	Description
OpenWindowTagList()	Open a window.
OpenWindowTags()	Alternate calling sequence for OpenWindowTagList().
OpenWindow()	Pre-V36 way to open a window.
CloseWindow()	Close a window.
BeginRefresh()	Turn on optimized window refresh mode.
EndRefresh()	Turn off optimized window refresh mode.
RefreshWindowFrame()	Redraw the borders and border gadgets of an open window.
ActivateWindow()	Make an open window active.
SizeWindow()	Change the size of an open window.
MoveWindow()	Change the position of an open window.
ChangeWindowBox()	Change the size and position of an open window.
WindowLimits()	Change the minimum and maximum sizes of an open window.
WindowToBack()	Move a window behind all other windows.
WindowToFront()	Move a window in front of all other windows.
MoveWindowInFrontOf()	Move a window in front of another window.
ZipWindow()	Change the size of window to its alternate size.
SetWindowTitles()	Change the window titles for the window and the screen.
SetPointer()	Set up a custom pointer to display whenever the window is active.
ClearPointer()	Restore the mouse pointer to its default imagery.

Chapter 5
INTUITION GADGETS

This chapter describes the multi-purpose software controls called gadgets. Gadgets are software controls symbolized by an image that the user can operate with the mouse or keyboard. They are the Amiga's equivalent of buttons, knobs and dials.

Much of the user's input to an application takes place through gadgets in the application's windows and requesters. Gadgets are also used by Intuition itself for handling screen and window movement and depth arrangement, as well as window sizing and closing.

Intuition maintains gadget imagery, watches for activation and deactivation and performs other management required by the gadget. The application can choose its level of involvement from simply receiving gadget activation messages to processing the actual mouse button presses and movements. To make gadget programming even easier, Release 2 of the Amiga operating system includes the new GadTools library. Applications written for Release 2 should take advantage of this new library (described separately in the ''GadTools Library'' chapter).

About Gadgets

There are two kinds of gadgets: *system gadgets* and *application gadgets*. System gadgets are set up by Intuition to handle the positioning and depth arranging of screens, and to handle the positioning, sizing, closing and depth arranging of windows. System gadgets always use the same imagery and location giving the windows and screens of any application a basic set of controls that are familiar and easy to operate. In general, applications do not have to do any processing for system gadgets; Intuition does all the work.

Application gadgets are set up by an application program. These may be the basic gadget types described in this chapter, the pre-fabricated gadgets supplied by the GadTools library, or special gadget types defined through Intuition's custom gadget and BOOPSI facilities. Application gadgets can be placed anywhere within a window and can use just about any image. The action associated with an application gadget is carried out by the application.

There are four basic types of application gadgets:

- Boolean (or button) gadgets elicit true/false or yes/no kinds of answers from the user.
- Proportional gadgets allow the user to select from a continuous range of options, such as volume or speed.
- String gadgets are used to get or display character based information (a special class of string gadget allows entry of numeric data.)
- Custom gadgets, a new, generalized form of gadget, provide flexibility to perform any type of function.

The way a gadget is used varies according to the type of gadget. For a boolean gadget, the user operates the gadget by simply clicking the mouse select button. For a string gadget, a cursor appears, allowing the user to enter data from the keyboard. For a proportional gadget, the user can either drag the knob with the mouse or click in the gadget container to move the knob by a set increment.

Gadgets are chosen by positioning the pointer within an area called the select box, which is application defined, and pressing the mouse select button (left mouse button).

When a gadget is selected, its imagery is changed to indicate that it is activated. The highlighting method for the gadget may be set by the application. Highlighting methods include alternate image, alternate border, a box around the gadget and color complementing.

A gadget can be either enabled or disabled. Disabled gadgets cannot be operated and are indicated by *ghosting* the gadget, that is, overlaying its image with a pattern of dots. Gadgets may also be directly modified and redrawn by first removing the gadget from the system.

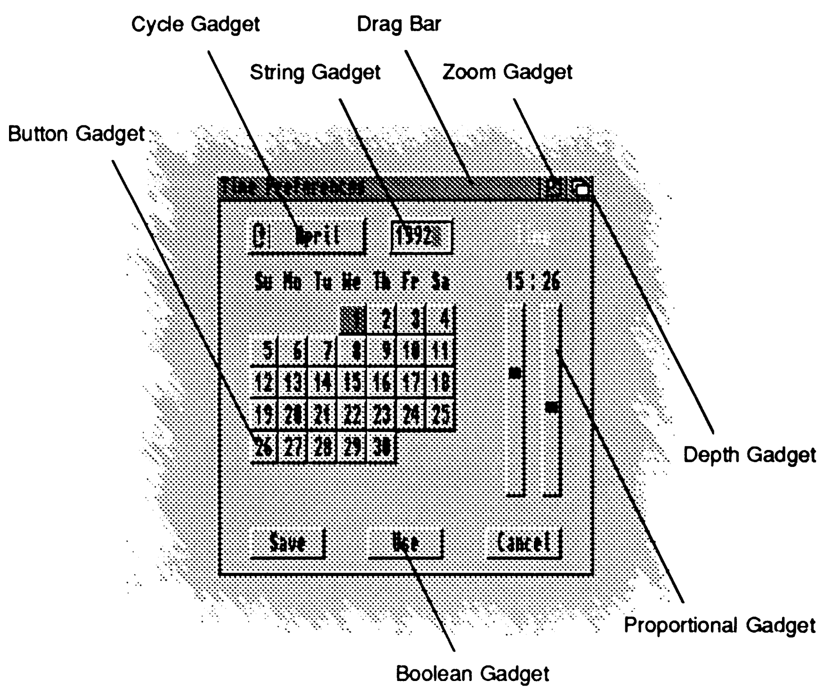


Figure 5-1: System and Application Gadgets

SYSTEM GADGETS

System gadgets are predefined gadgets provided by Intuition to support standard operations of windows and screens. System gadgets have a standard image and location in the borders of screens or windows. Intuition manages the operation of all system gadgets except the close gadget.

The drag and depth gadgets are automatically attached to each screen in the system. The application cannot control the creation of these gadgets, but can control their display and operation. Screens may be opened ''quiet'', without any of the gadget imagery displayed. Applications should avoid covering the screen's gadgets with windows as this may prevent the user from freely positioning the screen. See the ''Intuition Screens'' chapter for more information on the positioning and use of system gadgets for screens.

The drag, depth, close, sizing and zoom gadgets are available to be attached to each window. These gadgets are not provided automatically, the application must specify which gadgets it requires. See the ''Intuition Windows'' chapter for more information on the positioning and use of system gadgets for windows.

APPLICATION GADGETS

Application gadgets imitate real life controls: they are the switches, knobs, handles and buttons of the Intuition environment. Gadgets can be created with almost any imaginable type of imagery and function. Visual imagery for gadgets can combine text with hand drawn imagery or lines of multiple colors.

A gadget is created by declaring and initializing a **Gadget** structure as defined in *<intuition/intuition.h>*. See the ''Gadget Structure'' section later in this chapter for more details.

Gadgets always appear in a window or requester. All windows and requesters keep a list of the gadgets they contain. Gadgets can be added when the window or requester is opened, or they can be added or removed from the window or requester after it is open.

As with other types of input events, Intuition notifies your application about gadget activity by sending a message to your window's I/O channels: the IDCMP (**Window.UserPort**) or the console device. The message is sent as an Intuition **IntuiMessage** structure. The **Class** field of this structure is set to IDCMP_GADGETDOWN or IDCMP_GADGETUP with the **IAddress** field set to the address of the **Gadget** that was activated. (See the chapter on ''Intuition Input and Output Methods'' for details.)

Application gadgets can go anywhere in windows or requesters, including in the borders, and can be any size or shape. When application gadgets are placed into the window's border at the time the window is opened, Intuition will adjust the border size accordingly. Application gadgets are not supported in screens (although this may be simulated by placing the gadget in a backdrop window).

Gadget size can be fixed, or can change relative to the window size. Gadget position can be set relative to either the top or bottom border, and either the left or right border of the window, allowing the gadget to move with a border as the window is sized.

This flexibility provides the application designer the freedom to create gadgets that mimic real devices, such as light switches or joysticks, as well as the freedom to create controls that satisfy the unique needs of the application.

A Simple Gadget Example

The example below demonstrates a simple application gadget. The program declares a **Gadget** structure set up as a boolean gadget with complement mode highlighting. The gadget is attached to the window when it is opened by using the WA_Gadgets tag in the **OpenWindowTags()** call.

```
;/* simplegad.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 simplegad.c
Blink FROM LIB:c.o,simplegad.o TO simplegad LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**
** simplegad.c - show the use of a button gadget.
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library *IntuitionBase;

#define BUTTON_GAD_NUM   (3)
#define MYBUTTONGADWIDTH (100)
#define MYBUTTONGADHEIGHT (50)

/* NOTE that the use of constant size and positioning values are
** not recommended; it just makes it easy to show what is going on.
** The position of the gadget should be dynamically adjusted depending
** on the height of the font in the title bar of the window.
*/

UWORD buttonBorderData[] =
    {
    0,0, MYBUTTONGADWIDTH + 1,0, MYBUTTONGADWIDTH + 1,MYBUTTONGADHEIGHT + 1,
    0,MYBUTTONGADHEIGHT + 1, 0,0,
    };

struct Border buttonBorder =
    {
    -1,-1,1,0,JAM1,5,buttonBorderData,NULL,
    };

struct Gadget buttonGad =
    {
    NULL, 20,20, MYBUTTONGADWIDTH,MYBUTTONGADHEIGHT,
    GFLG_GADGHCOMP, GACT_RELVERIFY | GACT_IMMEDIATE,
    GTYP_BOOLGADGET, &buttonBorder, NULL, NULL,0,NULL,BUTTON_GAD_NUM,NULL,
    };

/*
** routine to show the use of a button (boolean) gadget.
*/
VOID main(int argc, char **argv)
{
struct Window *win;
struct IntuiMessage *msg;
struct Gadget *gad;
ULONG  class;
BOOL done;

/* make sure to get intuition version 37, for OpenWindowTags() */
IntuitionBase = OpenLibrary("intuition.library", 37);
```

```
if (IntuitionBase)
    {
    if (win = OpenWindowTags(NULL,
                            WA_Width, 400,
                            WA_Height, 100,
                            WA_Gadgets, &buttonGad,
                            WA_Activate, TRUE,
                            WA_CloseGadget, TRUE,
                            WA_IDCMP, IDCMP_GADGETDOWN | IDCMP_GADGETUP |
                                      IDCMP_CLOSEWINDOW,
                            TAG_END))
        {
        done = FALSE;
        while (done == FALSE)
            {
            Wait(1L << win->UserPort->mp_SigBit);
            while ( (done == FALSE) &&
                    (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
                {
                /* Stash message contents and reply, important when message
                ** triggers some lengthy processing
                */
                class = msg->Class;

                /* gadget address is ONLY valid for gadget messages! */
                if ((class == IDCMP_GADGETUP) || (class == IDCMP_GADGETDOWN))
                    gad = (struct Gadget *)(msg->IAddress);

                ReplyMsg((struct Message *)msg);

                /* switch on the type of the event */
                switch (class)
                    {
                    case IDCMP_GADGETUP:
                        /* caused by GACT_RELVERIFY */
                        printf("received an IDCMP_GADGETUP, gadget number %d\n",
                               gad->GadgetID);
                        break;
                    case IDCMP_GADGETDOWN:
                        /* caused by GACT_IMMEDIATE */
                        printf("received an IDCMP_GADGETDOWN, gadget number %d\n",
                               gad->GadgetID);
                        break;
                    case IDCMP_CLOSEWINDOW:
                        /* set a flag that we are done processing events... */
                        printf("received an IDCMP_CLOSEWINDOW\n");
                        done = TRUE;
                        break;
                    }
                }
            }
        CloseWindow(win);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

ADDING AND REMOVING GADGETS

Gadgets may be added to a window or requester when the window or requester is opened, or they may be added later. To add the gadgets when a window is opened, use the WA_Gadgets tag with the **OpenWindowTagList()** call. This technique is demonstrated in the example above. For a requester, set the **ReqGadget** field in the **Requester** structure to point to the first gadget in the list.

To add or remove gadgets in a window or requester that is already open, use **AddGList()** or **RemoveGList()**. These functions operate on gadgets arranged in a list. A gadget list is linked together by the **NextGadget** field of the **Gadget** structure (see the description of the **Gadget** structure later in this chapter).

AddGList() adds a gadget list that you specify to the existing gadget list of a window or requester:

```
UWORD AddGList( struct Window *window, struct Gadget *agadget,
                unsigned long position, long numGad, struct Requester *requester );
```

Up to **numGad** gadgets will be added from the gadget list you specify beginning with **agadget**. The **position** argument determines where your gadgets will be placed in the existing list of gadgets for the window or requester. Use (~0) to add your gadget list to the end of the window or requester's gadget list. This function returns the actual position where your gadgets are added in the existing list.

To remove gadgets from a window or requester use **RemoveGList()**:

```
UWORD RemoveGList( struct Window *remPtr, struct Gadget *agadget, long numGad );
```

This function removes up to **numGad** gadgets from a window or requester, beginning with the specified one. Starting with V37, if one of the gadgets that is being removed is the active gadget, this routine will wait for the user to release the mouse button before deactivating and removing the gadget. This function returns the former position of the removed gadget or -1 if the specified gadget was not found.

The **Gadget** structure should never be directly modified after it has been added to a window or requester. To modify a gadget, first remove it with **RemoveGList()**, modify the structure as needed, and then add the gadget back to the system with **AddGList()**. Finally, refresh the gadget imagery with **RefreshGList()**. (See the section on ''Gadget Refreshing'' below for more information.)

Some attributes of a gadget may be modified through special Intuition functions that perform the modification. When using such functions it is not necessary to remove, add or refresh the gadget. These functions, such as **NewModifyProp()**, **OnGadget()** and **OffGadget()**, are described later in this chapter.

Gadget Imagery

Gadget imagery can be rendered with a series of straight lines, a bitmap image or no imagery at all. In addition to the line or bitmap imagery, gadgets may include a series of text strings.

HAND DRAWN GADGETS

Bitmap or custom images are used as imagery for a gadget by setting the GFLG_GADGIMAGE flag in the **Flags** field of the **Gadget** structure. An **Image** structure must be set up to manage the bitmap data. The address of the **Image** structure is placed into the gadget's **GadgetRender** field. The bitmap image will be positioned relative to the gadget's select box. For more information about creating Intuition images, see the chapter ''Intuition Images, Line Drawing, and Text.'' For a listing of the **Gadget** structure and all its flags see the ''Gadget Structure'' section later in this chapter.

LINE DRAWN GADGETS

Gadget imagery can also be created by specifying a series of lines to be drawn. These lines can go around or through the select box of the gadget, and can be drawn using any color pen and draw mode. Multiple groups of lines may be specified, each with its own pen and draw mode.

The **Border** structure is used to describe the lines to be drawn. The **Border** structure is incorporated into the gadget by clearing the GFLG_GADGIMAGE flag in the gadget's **Flags** field. The address of the **Border** structure is placed into the gadget's **GadgetRender** field. The border imagery will be positioned relative to the gadget's select box. For more information about creating a **Border** structure, see the chapter "Intuition Images, Line Drawing, and Text."

GADGET TEXT

Gadgets may include text information in the form of a linked list of **IntuiText** structures. A pointer to the first **IntuiText** structure in the list is placed in the **Gadget** structure's **GadgetText** field. The text is positioned relative to the top left corner of the gadget's select box. For more information on **IntuiText**, see the "Intuition Images, Line Drawing and Text" chapter.

GADGETS WITHOUT IMAGERY

Gadgets can be created without any defining imagery. This type of gadget may be used where mouse information is required but graphical definition of the gadget is not, or where the existing graphics sufficiently define the gadget that no additional imagery is required. A gadget with no imagery may be created by clearing the GFLG_GADGIMAGE flag in the gadget's **Flags** field, and by setting the gadget's **GadgetRender** and **GadgetText** fields to NULL.

The text display of a word processor is a case where mouse information is required without any additional graphics. If a large gadget is placed over the text display, gadget and mouse event messages may be received at the IDCMP (**Window.UserPort**) when the mouse select button is either pressed or released. The mouse information is used to position the pointer in the text, or to allow the user to mark blocks of text. The drag bar of a window is another example of a gadget where existing imagery defines the gadget such that additional graphics are not required.

Gadget Selection

The user operates a gadget by pressing the select button while the mouse pointer is within the gadget's select box. Intuition provides two ways of notifying your program about the user operating a gadget. If your application needs immediate notification when the gadget is chosen, set the GACT_IMMEDIATE flag in the gadget's **Activation** field. Intuition will send an IDCMP_GADGETDOWN message to the window's **UserPort** when it detects the mouse select button being pressed on the gadget.

If the application needs notification when the gadget is released, i.e., when the user releases the mouse select button, set the GACT_RELVERIFY (for "release verify") flag in the gadget's **Activation** field. For boolean gadgets, Intuition will send an IDCMP_GADGETUP message to the window's **UserPort** when the mouse select button is released over a GACT_RELVERIFY gadget. The program will only receive the IDCMP_GADGETUP message if the user still has the pointer positioned over the select box of the gadget when the mouse select button is released.

If the user moves the mouse out of the gadget's select box before releasing the mouse button an IDCMP_MOUSEBUTTONS event will be sent with a code of SELECTUP. This indicates the user's desire to not proceed with the action. Boolean gadgets that are GACT_RELVERIFY allow the user a chance to cancel a selection by rolling the mouse off of the gadget before releasing the select button.

String gadgets have a slightly different behavior, in that they remain active after the mouse button has been released. The gadget remains active until Return or Enter is pressed, the user tabs to the next or previous gadget, another window becomes active or the user chooses another object with the mouse. An IDCMP_GADGETUP message is only sent for GACT_RELVERIFY string gadgets if the user ends the gadget interaction through the Return, Enter or (if activated) one of the tab keys.

GACT_RELVERIFY proportional gadgets send IDCMP_GADGETUP events even if the mouse button is released when the pointer is not positioned over the select box of the gadget.

Gadgets can specify both the GACT_IMMEDIATE and GACT_RELVERIFY activation types, in which case, the program will receive both IDCMP_GADGETDOWN and IDCMP_GADGETUP messages.

Gadget Size and Position

The position and dimensions of the gadget's select box are defined in the **Gadget** structure. The **LeftEdge**, **TopEdge**, **Width** and **Height** values can be absolute numbers or values relative to the size of the window. When using absolute numbers, the values are set once, when the gadget is created. When using relative numbers, the size and position of the select box are adjusted dynamically every time the window size changes.

The gadget image is positioned relative to the select box so when the select box moves the whole gadget moves. The size of the gadget image, however, is not usually affected by changes in the select box size (proportional gadgets are the exception). To create a gadget image that changes size when the select box and window change size, you have to handle gadget rendering yourself or use a BOOPSI gadget.

SELECT BOX POSITION

To specify relative position or size for the gadget's select box, set or more of the flags GFLG_RELRIGHT, GFLG_RELBOTTOM, GFLG_RELWIDTH or GFLG_RELHEIGHT in the **Flags** field of the **Gadget** structure. When using GFLG_RELxxx flags, the gadget size or position is recomputed each time the window is sized.

> *Positioning the Select Box.* With GFLG_RELxxx gadgets, all of the imagery must be contained within the gadget's select box. This allows Intuition to erase the gadget's imagery when the window is sized. Intuition must be able to erase the gadget's imagery since the gadget's position or size will change as the window size changes. If the old one were not removed, imagery from both sizes/positions would be visible.
>
> If a GFLG_RELxxx gadget's imagery must extend outside of its select box, position another GFLG_RELxxx gadget with a larger select box such that all of the first gadget's imagery is within the second gadget's select box. This ''shadow'' gadget is only used to clear the first gadget's imagery and, as such, it should not have imagery nor should it generate any messages. It should also be positioned later in the gadget list than the first gadget so that its select box does not interfere with the first gadget.

The left-right position of the select box is defined by the variable **LeftEdge**, which is an offset from either the left or right edge of the display element. The offset method is determined by the GFLG_RELRIGHT flag. For the **LeftEdge** variable, positive values move toward the right and negative values move toward the left of the containing display element. If GFLG_RELRIGHT is cleared, **LeftEdge** is an offset (usually a positive value) from the left edge of the display element.

If GFLG_RELRIGHT is set, **LeftEdge** is an offset (usually a negative value) from the right edge of the display element. When this is set, the left-right position of the select box in the window is recomputed each time the window is sized. The gadget will automatically move with the left border as the window is sized.

The top-bottom position of the select box is defined by the variable **TopEdge**, which is an offset from either the top or bottom edge of the display element (window or requester). The offset method is determined by the GFLG_RELBOTTOM flag. For the **TopEdge** variable, positive values move toward the bottom and negative values move toward the top of the containing display element.

If GFLG_RELBOTTOM is cleared, **TopEdge** is an offset (usually a positive value) from the top of the display element. If GFLG_RELBOTTOM is set, **TopEdge** is an offset (usually a negative value) from the bottom of the display element. When this is set, the position of the select box is recomputed each time the window is sized. The gadget will automatically move with the bottom border as the window is sized.

SELECT BOX DIMENSION

The height and width of the gadget select box can be absolute or they can be relative to the height and width of the display element in which the gadget resides.

Set the gadget's GFLG_RELWIDTH flag to make the gadget's width relative to the width of the window. When this flag is set, the **Width** value is added to the current window width to determine the width of the gadget select box. The **Width** value is usually negative in this case, making the width of the gadget smaller than the width of the window. If GFLG_RELWIDTH is not set, **Width** will specify the actual width of the select box.

The actual width of the box will be recomputed each time the window is sized. Setting GFLG_RELWIDTH and a gadget width of zero will create a gadget that is always as wide as the window, regardless of how the window is sized.

The GFLG_RELHEIGHT flag has the same effect on the height of the gadget select box. If the flag is set, the height of the select box will be relative to the height of the window, and the actual height will be recomputed each time the window is sized. If the flag is not set, the value will specify the actual height of the select box.

Here are a few examples of gadgets that take advantage of the special relativity modes of the select box. Consider the Intuition window sizing gadget. The **LeftEdge** and **TopEdge** of this gadget are both defined relative to the right and bottom edges of the window. No matter how the window is sized, the gadget always appears in the lower right corner.

For the window drag gadget, the **LeftEdge** and **TopEdge** are always absolute in relation to the top left corner of the window. **Height** of this gadget is always an absolute quantity. **Width** of the gadget, however, is defined to be zero. When **Width** is combined with the effect of the GFLG_RELWIDTH flag, the drag gadget is always as wide as the window.

For a program with several requesters, each of which has an "OK" gadget in the lower left corner and a "Cancel" gadget in the lower right corner, two gadgets may be designed that will appear in the correct position regardless of the size of the requester. Design the "OK" and "Cancel" gadgets such that they are relative to the lower left and lower right corners of the requester. Regardless of the size of the requesters, these gadgets will appear in the correct position relative to these corners. Note that these gadgets may only be used in one window or requester at a time.

POSITIONING GADGETS IN WINDOW BORDERS

Gadgets may be placed in the borders of a window. To do this, set one or more of the border flags in the **Gadget** structure and position the gadget in the window border. Setting these flags also tells Intuition to adjust the size of the window's borders to accommodate the gadget.

Borders are adjusted only when the window is opened. Although the application can add and remove gadgets with **AddGList()** and **RemoveGList()** after the window is opened, Intuition does not readjust the borders.

A gadget may be placed into two borders by setting multiple border flags. If a gadget is to be placed in two borders, it only makes sense to put the gadget into adjoining borders. Setting both side border flags or both the top and bottom border flags for a particular gadget, will create a window that is all border.

See the SuperBitMap example, lines.c, in the "Intuition Windows" chapter for an example of creating proportional gadgets that are positioned within a window's borders.

There are circumstances where the border size will not adjust properly so that the gadget has the correct visual appearance. One way to correct this problem is to place a "hidden" gadget into the border, which forces the border to the correct size. Such a gadget would have no imagery and would not cause any IDCMP messages to be sent on mouse button activity. The gadget should be placed at the end of the gadget list of the window, so that it does not cover up any other border gadgets.

Sometimes the sizing gadget can be used to adjust the width of the borders, as in the case of proportional gadgets in the right or bottom border. The proportional gadget will only increase the width of the border by enough so that the select box of the gadget fits within the border, no space is reserved between the gadget and the inner edge of the window. By placing the size gadget in both borders (using the window flags WFLG_SIZEBRIGHT and WFLG_SIZEBBOTTOM), the prop gadget sizes can be adjusted so that there is an even margin on all sides. This technique is used in the lines.c example mentioned above.

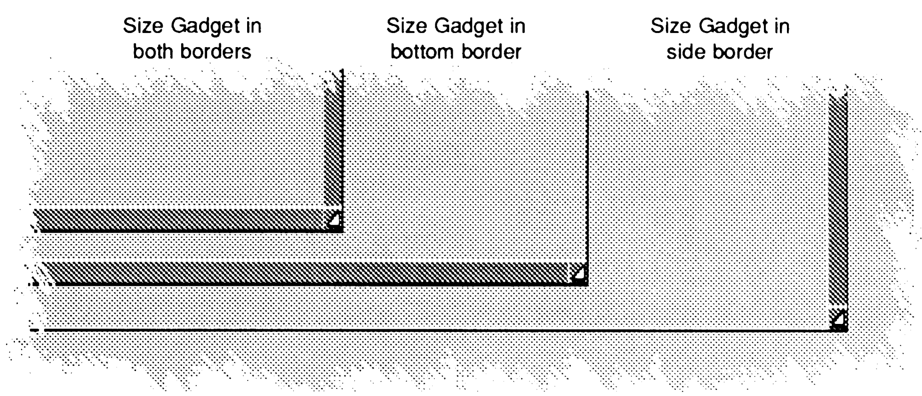


Figure 5-2: Size Gadget in Various Window Border Combinations

The border flags GACT_RIGHTBORDER, GACT_LEFTBORDER, GACT_TOPBORDER and GACT_BOTTOMBORDER which are set in the **Activation** field of the **Gadget** structure declare that the gadget will be positioned in the border. Gadgets which are declared to be in the border are automatically refreshed by Intuition whenever the window borders need to be redrawn. This prevents the gadget imagery from being obliterated.

Some applications forgot to declare some of their gadgets as being in the border. While they looked fine prior to V36, they would have had some gadget imagery overwritten by the new style of window borders introduced with that release. To ensure compatibility with such applications, Intuition attempts to identify gadgets that are substantially in the border but do not have the appropriate border flags set. Such gadgets are marked for the same refresh treatment as explicit border gadgets. Applications should not rely on this behavior, but should instead declare their border gadgets properly.

Gadgets that are not declared to be in the border, and whose dimensions are 1 x 1 or smaller are never marked by Intuition as being effectively in the border. This is because some applications tuck a small non-selectable gadget (of size 0x0 or 1x1) into the window border, and attach imagery for the window to that gadget. The application does this to get automatic refresh of that imagery, since Intuition refreshes gadget imagery when window damage occurs.

Beginning with V36, Intuition attempts to locate gadgets within the border that do not have the appropriate flags set. This may cause gadgets to appear in the border when the application has not set any of the border flags. Applications should not rely on this behavior, nor should they place non-border gadgets fully or partially within the window's borders.

Gadget Highlighting

In general, the appearance of an active or selected gadget changes to inform the user the gadget state has changed. A highlighting method is specified by setting one of the highlighting flags in the **Gadget** structure's **Flags** field.

Intuition supports three methods of activation or selection highlighting:

- Highlighting by color complementing (GFLG_GADGHCOMP)
- Highlighting by drawing a box (GFLG_GADGHBOX)
- Highlighting by an alternate image or border (GFLG_GADGHIMAGE)
- No highlighting (GFLG_GADGHNONE)

One of the highlighting types or GFLG_GADGHNONE must be specified for each gadget.

HIGHLIGHTING BY COLOR COMPLEMENTING

Highlighting may be accomplished by complementing all of the colors in the gadget's select box. In this context, complementing means the complement of the binary number used to represent a particular color register. For example, if the color in color register 2 is used (binary 10) in a specific pixel of the gadget, the complemented value of that pixel will be the color in color register 1 (binary 01).

To use this highlighting method, set the GFLG_GADGHCOMP flag.

Only the select box of the gadget is complemented; any portion of the text, image, or border which is outside of the select box is not disturbed. See the chapter "Intuition Images, Line Drawing, and Text," for more information about complementing and about color in general.

HIGHLIGHTING BY DRAWING A BOX

To highlight by drawing a simple border around the gadget's select box, set the GFLG_GADGHBOX bit in the **Flags** field.

HIGHLIGHTING WITH AN ALTERNATE IMAGE OR ALTERNATE BORDER

An alternate image may be supplied as highlighting for gadgets that use image rendering, similarly an alternate border may be supplied for gadgets that use border rendering. When the gadget is active or selected, the alternate image or border is displayed in place of the default image or border. For this highlighting method, set the **SelectRender** field of the **Gadget** structure to point to the **Image** structure or **Border** structure for the alternate display.

Specify that highlighting is to be done with alternate imagery by setting the GFLG_GADGHIMAGE flag in the **Flags** field of the **Gadget** structure. When using GFLG_GADGHIMAGE, remember to set the GFLG_GADGIMAGE flag for images, clear it for borders.

When using alternate images and borders for highlighting, gadgets rendered with images must highlight with another image and gadgets rendered with borders must highlight with another border. For information about how to create an **Image** or **Border** structure, see the chapter ''Intuition Images, Line Drawing, and Text.''

Gadget Refreshing

Gadget imagery is redrawn by Intuition at appropriate times, e.g., when the user operates the gadget. The imagery can also be updated under application control.

GADGET REFRESHING BY INTUITION

Intuition will refresh a gadget whenever an operation has damaged the layer of the window or requester to which it is attached. Because of this, the typical application does not need to call **RefreshGList()** as a part of its IDCMP_REFRESHWINDOW event handling.

Intuition's refreshing of the gadgets of a damaged layer is done through the layer's damage list. This means that rendering is clipped or limited to the layer's damage region--that part of the window or requester that needs refreshing.

Intuition directly calls the Layers library functions **BeginUpdate()** and **EndUpdate()**, so that rendering is restricted to the proper area. Applications should not directly call these functions under Intuition, instead, use the **BeginRefresh()** and **EndRefresh()** calls. Calls to **RefreshGList()** or **RefreshGadgets()** between **BeginRefresh()** and **EndRefresh()** are not permitted. Never add or remove gadgets between the **BeginRefresh()** and **EndRefresh()** calls.

For more information on **BeginRefresh()** and **EndRefresh()**, see the ''Intuition Windows'' chapter and the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Gadgets which are positioned using GFLG_RELBOTTOM or GFLG_RELRIGHT, or sized using GFLG_RELWIDTH or GFLG_RELHEIGHT pose a problem for this scheme. When the window is sized,

the images for these gadgets must change, even though they are not necessarily in the damage region. Therefore, Intuition must add the original and new visual regions for such relative gadgets to the damage region before refreshing the gadgets. The result of this is that applications should ensure that any gadgets with relative position or size do not have **Border, Image** or **IntuiText** imagery that extends beyond their select boxes.

GADGET REFRESHING BY THE PROGRAM

The **AddGList()** function adds gadgets to Intuition's internal lists but do not display their imagery. Subsequently calls to **RefreshGList()** must be made to draw the gadgets into the window or requester.

Programs may use **RefreshGList()** to update the display after making changes to their gadgets. The supported changes include (not an exhaustive list): changing the GFLG_SELECTED flag for boolean gadgets to implement mutually exclusive gadgets, changing the **GadgetText** of a gadget to change its label, changing the GFLG_DISABLED flag, and changing the contents of the **StringInfo** structure **Buffer** of a string gadget. When making changes to a gadget, be sure to remove the gadget from the system with **RemoveGList()** before altering it. Remember to add the gadget back and refresh its imagery.

Boolean gadgets rendered with borders, instead of images, or highlighted with surrounding boxes (GFLG_GADGHBOX) are handled very simply by Intuition, and complicated transitions done by the program can get the rendering out of phase. Applications should avoid modifying the imagery and refreshing gadgets that may be highlighted due to selection by the user. Such operations may leave pixels highlighted when the gadget is no longer selected. The problems with such transitions can often be avoided by providing imagery, either image or border, that covers all pixels in the select box. For GFLG_GADGHIMAGE gadgets, the select imagery should cover all pixels covered in the normal imagery.

Updating a Gadget's Imagery

The **RefreshGList()** function was designed to draw gadgets from scratch, and assumes that the underlying area is blank. This function cannot be used blindly to update gadget imagery. The typical problem that arises is that the application cannot change a gadget from selected to unselected state (or from disabled to enabled state) and have the imagery appear correct. However, with a little care, the desired results can be obtained.

Depending on the imagery you select for your gadget, the rendering of one state may not completely overwrite the rendering of a previous one. For example, consider a button which consists of a complement-highlighted boolean gadget, whose imagery is a surrounding **Border** and whose label is an **IntuiText**. Attempting to visually unselect such a gadget by clearing its GFLG_SELECTED flag and refreshing it will leave incorrect imagery because **RefreshGList()** just redraws the border and text, and never knows to erase the field area around the text and inside the gadget. That area will remain complemented from before.

One solution is to use a gadget whose imagery is certain to overwrite any imagery left over from a different state. Disabling a gadget or highlighting it with complement mode affects the imagery in the entire select box. To overwrite this successfully, the gadget's imagery (**GadgetRender**) should be an **Image** structure which fully covers the select box. Such a gadget may be highlighted with color complementing (GFLG_GADGHCOMP), or with an alternate image (GFLG_GADGHIMAGE) for its **SelectRender**. Or, for a gadget which will never be disabled but needs to be deselected programmatically, you may also use a **Border** structure for its **GadgetRender**, and an identically-shaped (but differently colored) **Border** for its **SelectRender**.

The other technique is to pre-clear the underlying area before re-rendering the gadget. To do this, remove the gadget, erase the rectangle of the gadget's select area, change the GFLG_SELECTED or the GFLG_DISABLED flag, add the gadget back, and refresh it.

If the gadget has a relative size and/or position (i.e., if of the GFLG_RELxxx flags are used), then the application will need to compute the rectangle of the gadget's select area based on the window's current width and/or height. Since the window size is involved in the calculation, it is important that the window not change size between the call to **RemoveGList()** and the call to **RectFill()**. To ensure this, the application should set IDCMP_SIZEVERIFY so that Intuition will first notify you before beginning a sizing operation. (Note that applications using any of the IDCMP verify events such as IDCMP_SIZEVERIFY should not delay long in processing such events, since that holds up the user, and also Intuition may give up and stop waiting for you).

Gadget Refresh Function

Use the **RefreshGList()** function to refresh one or more gadgets in a window or requester.

```
void RefreshGList( struct Gadget *gadgets, struct Window *window,
    struct Requester *requester, long numGad );
```

This function redraws no more than **numGad** gadgets, starting with the specified gadget, in a window or requester. The application should refresh any gadgets after adding them. The function should also be used after the application has modified the imagery of the gadgets to display the new imagery.

Gadget Enabling and Disabling

A gadget may be disabled so that it cannot be chosen by the user. When a gadget is disabled, its image is *ghosted.* A ghosted gadget is overlaid with a pattern of dots, thereby making the imagery less distinct. The dots are drawn into the select box of the gadget and any imagery that extends outside of the select box is not affected by the ghosting.

The application may initialize whether a gadget is disabled by setting the GFLG_DISABLED flag in the **Gadget** structure's **Flags** field before a gadget is submitted to Intuition. Clear this flag to create an enabled gadget.

After a gadget is submitted to Intuition for display, its current enable state may be changed by calling **OnGadget()** or **OffGadget()**. If the gadget is in a requester, the requester must currently be displayed when calling these functions.

```
void OnGadget ( struct Gadget *gadget, struct Window *window, struct Requester *requester );
void OffGadget( struct Gadget *gadget, struct Window *window, struct Requester *requester );
```

Depending on what sort of imagery you choose for your gadget, **OnGadget()** may not be smart enough to correct the gadget's displayed imagery. See the section on "Updating a Gadget's Imagery" for more details.

Multiple gadgets may be enabled or disabled by calling **OnGadget()** or **OffGadget()** for each gadget, or by removing the gadgets with **RemoveGList()**, setting or clearing the GFLG_DISABLED flag on each, replacing the gadgets with **AddGList()**, and refreshing with **RefreshGList()**.

Gadget Pointer Movements

If the GACT_FOLLOWMOUSE flag is set for a gadget, the application will receive mouse movement broadcasts as long as the gadget is active. This section covers the behavior of proportional, boolean and string gadgets, although there are major caveats in some cases:

- Unlike IDCMP_GADGETUP and IDCMP_GADGETDOWN **IntuiMessages**, the **IAddress** field of an IDMP_MOUSEMOVE **IntuiMessage** does not point to the gadget. The application must track the active gadget (this information is readily obtained from the IDCMP_GADGETDOWN message) instead of using the **IAddress** field.

  Right

  ```
  imsg=GetMsg(win->UserPort);
  class=imsg->Class;
  code=imsg->Code;
  /* OK */
  iaddress=imsg->IAddress;
  ReplyMsg(imsg);
  ```

  Wrong

  ```
  imsg=GetMsg(win->UserPort);
  class=imsg->Class;
  code=imsg->Code;
  /* ILLEGAL ! */
  gadid=((struct Gadget *)imsg->IAddress)->GadgetID;
  ReplyMsg(imsg);
  ```

  Using the code in the left column, it is acceptable to get the address of a gadget with `gadid=((struct Gadget *)iaddress)->GadgetID` but *only after* you have checked to make sure the message is an IDCMP_GADGETUP or IDCMP_GADGETDOWN.

- Boolean gadgets only receive mouse messages if both GACT_RELVERIFY and GACT_FOLLOWMOUSE are set. Those cases described below with GACT_RELVERIFY cleared do not apply to boolean gadgets.

- In general, IDCMP_MOUSEMOVE messages are sent when the mouse changes position while the gadget is active. Boolean and proportional gadgets are active while the mouse button is held down, thus mouse move messages will be received when the user "drags" with the mouse. String gadgets are active until terminated by keyboard entry or another object becomes active (generally by user clicking the other object). GACT_FOLLOWMOUSE string gadgets will generate mouse moves the entire time they are active, not just when the mouse button is held.

The broadcasts received differ according to the gadget's flag settings. If using the GACT_IMMEDIATE and GACT_RELVERIFY activation flags, the program gets a gadget down message, receives mouse reports (IDCMP_MOUSEMOVE) as the mouse moves, and receives a gadget up message when the mouse button is released. For boolean gadgets, the mouse button must be released while the pointer is over the gadget. If the button is not released over the boolean gadget, an IDCMP_MOUSEBUTTONS message with the SELECTUP qualifier will be sent.

If only using the GACT_IMMEDIATE activation flag, the program gets a gadget down message and receives mouse reports as the mouse moves. The mouse reports will stop when the user releases the mouse select button. This case does not apply to boolean gadgets as GACT_RELVERIFY must be set for boolean gadgets to receive mouse messages. If only using the GACT_RELVERIFY activation flag, the program gets mouse reports followed by an up event for a gadget. For boolean gadgets, the IDCMP_GADGETUP event will only be received if the button was released while the pointer was over the gadget. If the button is not released over the boolean gadget, a IDCMP_MOUSEBUTTONS message with the SELECTUP qualifier will be received if the program is receiving these events.

If neither the GACT_IMMEDIATE nor the GACT_RELVERIFY activation flags are set, the program will only receive mouse reports. This case does not apply to boolean gadgets as GACT_RELVERIFY must be set for boolean gadgets to receive mouse messages.

Gadget Structure

Here is the specification for the **Gadget** structure defined in <intuition/intuition.h>. You create an instance of this structure for each gadget you place in a window or requester:

```
struct Gadget
    {
    struct Gadget *NextGadget;
    WORD LeftEdge, TopEdge;
    WORD Width, Height;
    UWORD Flags;
    UWORD Activation;
    UWORD GadgetType;
    APTR GadgetRender;
    APTR SelectRender;
    struct IntuiText *GadgetText;
    LONG MutualExclude;
    APTR SpecialInfo;
    UWORD GadgetID;
    APTR UserData;
    };
```

NextGadget

Applications may create lists of gadgets that may be added to a window or requester with a single instruction. **NextGadget** is a pointer to the next gadget in the list. The last gadget in the list should have a **NextGadget** value of NULL.

When gadgets are added or removed, Intuition will modify the appropriate **NextGadget** fields to maintain a correctly linked list of gadgets for that window or requester. However, removing one or more gadgets does not reset the last removed gadget's **NextGadget** field to NULL.

LeftEdge, TopEdge, Width, Height

These variables describe the location and dimensions of the select box of the gadget. Both location and dimensions can be either absolute values or else relative to the edges and size of the window or requester that contains the gadget.

LeftEdge and **TopEdge** are relative to one of the corners of the display element, according to how GFLG_RELRIGHT and GFLG_RELBOTTOM are set in the **Flags** variable (see below).

Width and **Height** are either absolute dimensions or a negative increment to the width abd height of a requester or a window, according to to how the GFLG_RELWIDTH and GFLG_RELHEIGHT flags are set (see below).

Flags

The **Flags** field is shared by the program and Intuition. See the section below on ''Gadget Flags'' for a complete description of all the flag bits.

Activation

This field is used for information about some gadget attributes. See the ''Gadget Activation Flags'' section below for a description of the various flags.

GadgetType

This field contains information about gadget type and in what sort of display element the gadget is to be displayed. One of the following flags must be set to specify the type:

GTYP_BOOLGADGET

Boolean gadget type.

GTYP_STRGADGET

String gadget type. For an integer gadget, also set the GACT_LONGINT flag. See the "Gadget Activation Flags" section below.

GTYP_PROPGADGET

Proportional gadget type.

GTYP_CUSTOMGADGET

Normally not set by the application. Used by custom BOOPSI gadget types, discussed in the "BOOPSI" chapter.

The following gadget types may be set in addition to one of the above types. None of the following types are required:

GTYP_GZZGADGET

If the gadget is placed in a GimmeZeroZero window, setting this flag will place the gadget in the border layer, out of the inner window. If this flag is not set, the gadget will go into the inner window. Do not set this bit if this gadget is not placed in a GimmeZeroZero window.

GTYP_REQGADGET

Set this bit if this gadget is placed in a requester.

GadgetRender

A pointer to the **Image** or **Border** structure containing the graphics imagery of this gadget. If this field is set to NULL, no rendering will be done.

If the graphics of this gadget are implemented with an **Image** structure, this field should contain a pointer to that structure and the GFLG_GADGIMAGE flag must be set. If a **Border** structure is used, this field should contain a pointer to the **Border** structure, and the GFLG_GADGIMAGE bit must be cleared.

SelectRender

If the application does not use an alternate image for highlighting, set this field to NULL. Otherwise, if the flag GFLG_GADGHIMAGE is set, this field must contain a pointer to an **Image** or **Border** structure. The GFLG_GADGIMAGE flag determines the type of the rendering. Provide a pointer to an **IntuiText** structure to include a text component to the gadget. Multiple **IntuiText** structures may be chained. Set this field to NULL if the gadget has no associated text.

GadgetText

Provide a pointer to an **IntuiText** structure to include a text component to the gadget. Multiple **IntuiText** structures may be chained. Set this field to NULL if the gadget has no associated text.

The offsets in the **IntuiText** structure are relative to the top left of the gadget's select box.

MutualExclude

This field is currently ignored by Intuition, but is reserved. Do not store information here. Starting with V36, if the GadgetType is GTYP_CUSTOMGADGET this field is used to point to a Hook for the custom gadget.

SpecialInfo

SpecialInfo contains a pointer to an extension structure which contains the special information needed by the gadget.

If this is a proportional gadget, this variable must contain a pointer to an instance of a **PropInfo** data structure. If this is a string or integer gadget, this variable must contain a pointer to an instance of a **StringInfo** data structure. If this is a boolean gadget with GACT_BOOLEXTEND activation, this variable must contain a pointer to an instance of a **BoolInfo** data structure. Otherwise, this variable is ignored.

GadgetID

This variable is for application use and may contain any value. It is often used to identify the specific gadget within an event processing loop. This variable is ignored by Intuition.

UserData

This variable is for application use and may contain any value. It is often used as a pointer to a data block specific to the application or gadget. This variable is ignored by Intuition.

GADGET FLAGS

The following are the flags that can be set in the **Flags** variable of the **Gadget** structure. There are four highlighting methods to choose from. These determine how the gadget imagery will be changed when the gadget is selected. One of these four flags *must* be set.

GFLG_GADGHNONE

Set this flag for no highlighting.

GFLG_GADGHCOMP

This flag chooses highlighting by complementing all of the bits contained within the gadget's select box.

GFLG_GADGHBOX

This flag chooses highlighting by drawing a complemented box around the gadget's select box.

GFLG_GADGHIMAGE

Set this flag to indicate highlighting with an alternate image.

In addition to the highlighting flags, these other values may be set in the **Flags** field of the **Gadget** structure.

GFLG_GADGIMAGE

If the gadget has a graphic, and it is implemented with an **Image** structure, set this bit. If the graphic is implemented with a **Border** structure, make sure this bit is clear. This bit is also used by **SelectRender** to determine the rendering type.

GFLG_RELBOTTOM

Set this flag if the gadget's **TopEdge** variable describes an offset relative to the bottom of the display element (window or requester) containing it. A GFLG_RELBOTTOM gadget moves automatically as its window is made taller or shorter. Clear this flag if **TopEdge** is relative to the top of the display element. If GFLG_RELBOTTOM is set, **TopEdge** should contain a negative value, which will position it up from the bottom of the display element.

GFLG_RELRIGHT

Set this flag if the gadget's **LeftEdge** variable describes an offset relative to the right edge of the display element containing it. A GFLG_RELRIGHT gadget moves automatically as its window is made wider or narrower. Clear this flag if **LeftEdge** is relative to the left edge of the display element. If GFLG_RELRIGHT is set, **LeftEdge** should contain a negative value, which will position the gadget left of the right edge of the display element.

GFLG_RELWIDTH

Set this flag for "relative gadget width." If this flag is set, the width of the gadget's select box changes automatically whenever the width of its window changes. When using GFLG_RELWIDTH, set the gadget's **Width** to a negative value. This value will be added to the width of the gadget's display element (window or requester) to determine the actual width of the gadget's select box.

GFLG_RELHEIGHT

Set this flag for "relative gadget height." If this flag is set, the height of the gadget's select box changes automatically whenever the height of its window changes. When using GFLG_RELHEIGHT, set the gadget's **Height** to a negative value. This value will be added to the height of the gadget's display element (window or requester) to determine the actual height of the gadget's select box.

GFLG_SELECTED

Use this flag to preset the on/off selected state for a toggle-select boolean gadget (see the discussion of the GACT_TOGGLESELCT flag below). If the flag is set, the gadget is initially selected and is highlighted. If the flag is clear, the gadget starts off in the unselected state. To change the selection state of one or more gadgets, change their GFLG_SELECTED bits as appropriate, add them back and refresh them. However, see the section on "Updating a Gadget's Imagery" for important details.

GFLG_DISABLED

If this flag is set, this gadget is disabled. To enable or disable a gadget after the gadget has been added to the system, call the routines **OnGadget()** and **OffGadget()**. The GFLG_DISABLED flag can be programmatically altered in much the same way as GFLG_SELECTED above. See the section on "Updating a Gadget's Imagery" for important details.

GFLG_STRINGEXTEND

The **StringInfo Extension** field points to a valid **StringExtend** structure. Use of this structure is described later in the "String Gadget Type" section of this chapter. This flag is ignored prior to V37, see GACT_STRINGEXTEND for the same functionality under V36. Note that GACT_STRINGEXTEND is not ignored prior to V36 and should only be set in V36 or later systems.

GFLG_TABCYCLE

This string participates in cycling activation with the tab (or shifted tab) key. If this flag is set, the tab keys will de-activate this gadget as if the Return or Enter keys had been pressed, sending an IDCMP_GADGETUP message to the application, then the next string gadget with GFLG_TABCYCLE set will be activated. Shifted tab activates the previous gadget.

GADGET ACTIVATION FLAGS

These flags may be set in the **Activation** field of the **Gadget** structure.

GACT_TOGGLESELECT
: This flag applies only to boolean gadgets, and tells Intuition that this is to be a toggle-select gadget, not a hit-select one. Preset the selection state with the gadget flag GFLG_SELECTED (see above). The program may check if the gadget is in the selected state by examining the GFLG_SELECTED flag at any time.

GACT_IMMEDIATE
: If this bit is set, the program will be sent an IDCMP_GADGETDOWN message when the gadget is first picked. The message will be sent when the user presses the mouse select button.

GACT_RELVERIFY
: This is short for "release verify." If this bit is set, the program will be sent an IDCMP_GADGETUP message when the gadget is deactivated. IDCMP_GADGETUP will be sent for boolean gadgets when the user releases the mouse select button while the pointer is over the select box, for proportional gadgets whenever the user releases the mouse select button (regardless of the pointer position), and for string and integer gadgets when the user completes the text entry by pressing return or tabbing to the next gadget (where supported).

  For boolean gadgets, if the user releases the mouse button while the pointer is outside of the gadget's select box IDCMP_GADGETUP will not be generated. Instead, the program will receive an IDCMP_MOUSEBUTTONS event with the SELECTUP code set. For string gadgets, if the user deactivates the gadget by clicking elsewhere, it may not be possible to detect.

GACT_ENDGADGET
: This flag pertains only to gadgets attached to requesters. If a gadget with the GACT_ENDGADGET flag set is chosen by the user the requester will be terminated as if the application had called the **EndRequest()** function.

  See the chapter "Intuition Requesters and Alerts," for more information about requester gadget considerations.

GACT_FOLLOWMOUSE
: These flags may be set in the **Activation** field of the **Gadget** structure. As long as a gadget that has this flag set is active, the program will receive mouse position messages for each change of mouse position. For GTYP_BOOLGADGET gadgets, GACT_RELVERIFY must also be set for the program to receive mouse events.

The following flags are used to place application gadgets into a specified window border. Intuition will adjust the size of a window's borders appropriately provided these gadgets are set up with a call to **OpenWindow()**, **OpenWindowTags()** or **OpenWindowTagList()**. Intuition knows to refresh gadgets marked with these flags when the window border is changed, e.g., when the window is activated. For GimmeZeroZero windows, the GTYP_GZZGADGET flag must also be set for border gadgets.

GACT_RIGHTBORDER
: If this flag is set, the gadget is placed in the right border of the window and the width and position of this gadget are used in deriving the width of the window's right border.

GACT_LEFTBORDER
If this flag is set, the gadget is placed in the left border of the window and the width and position of this gadget are used in deriving the width of the window's left border.

GACT_TOPBORDER
If this flag is set, the gadget is placed in the top border of the window and the height and position of this gadget are used in deriving the height of the window's top border.

GACT_BOTTOMBORDER
If this flag is set, the gadget is placed in the bottom border of the window and the height and position of this gadget are used in deriving the height of the window's bottom border.

The following flags apply only to string gadgets:

GACT_STRINGCENTER
If this flag is set, the text in a string gadget is centered within the select box.

GACT_STRINGRIGHT
If this flag is set, the text in a string gadget is right justified within the select box.

GACT_STRINGLEFT
This "flag" has a value of zero. By default, the text in a string gadget is left justified within the select box.

GACT_LONGINT
If this flag is set, the user can construct a 32-bit signed integer value in a normal string gadget. The input buffer of the string gadget must be initialized with an ASCII representation of the starting integer value.

GACT_ALTKEYMAP
These flags may be set in the **Activation** field of the **Gadget** structure. A pointer to the keymap must be placed in the **StringInfo** structure variable **AltKeyMap**.

GACT_BOOLEXTEND
This flag applies only to boolean gadgets. If this flag is set, then the boolean gadget has a **BoolInfo** structure associated with it. A pointer to the **BoolInfo** structure must be placed in the **SpecialInfo** field of the **Gadget** structure.

GACT_STRINGEXTEND
This is an obsolete flag originally defined in V36. It applies only to string gadgets and indicates that **StringInfo.Extension** points to a valid **StringExtend** structure. Although this flag works, it is not ignored prior to V36 as it should be in order to be backward compatible. This flag is replaced by GFLG_STRINGEXTEND in V37. GFLG_STRINGEXTEND performs the same function and is properly ignored on systems prior to V36.

Boolean Gadget Type

A boolean gadget gets yes/no or on/off responses from the user. To make a boolean gadget set the **GadgetType** field to GTYP_BOOLGADGET in the **Gadget** structure.

Boolean gadgets come in two types: hit-select and toggle-select. Hit-select gadgets are only active while the user holds down the mouse select button. When the button is released, the gadget is unhighlighted. Action buttons, such as "OK" and "Cancel", are hit-select.

Toggle-select gadgets become selected when the user clicks them. To "unselect" the gadget, the user has to click the gadget again. Switches, such as a checkbox, are toggle-select.

Set the GACT_TOGGLESELECT flag in the **Activation** field of the **Gadget** structure to create a toggle-select gadget.

The GFLG_SELECTED flag in **Gadget** structure **Flags** field determines the initial and current on/off selected state of a toggle-select gadget. If GFLG_SELECTED is set, the gadget will be highlighted. The application can set the GFLG_SELECTED flag before submitting the gadget to Intuition. The program may examine this flag at any time to determine the current state of this gadget.

Try to make the imagery for toggle-select gadgets visually distinct from hit-select gadgets so that their operation can be determined by the user through visual inspection.

MASKED BOOLEAN GADGETS

Imagery for boolean gadgets is rectangular by default, but non-rectangular boolean gadgets are possible, with some restrictions. An auxiliary bit plane, called a mask, may be associated with a boolean gadget. When the user clicks within the select box of the gadget, a further test is made to see if the chosen point is contained within the mask. Only if it is, does the interaction count as a gadget hit.

With masked boolean gadgets, if the gadget has highlight type GFLG_GADGHCOMP then the complement rendering is restricted to the mask. This allows for non-rectangular shapes, such as an oval gadget which highlights only within the oval.

There are some shortcomings to non-rectangular boolean gadgets. For instance, the gadget image is not rendered through the mask. Images are rectangular blocks, with all bits rendered. In the case of an oval mask, the image will be rendered in the corner areas even though they are outside of the oval. Also, it is not possible to mask out the select box, thus non-rectangular masked gadgets cannot overlap in the masked area. Therefore, such gadgets can't be crowded together without care. Likewise, the ghosting of a disabled gadget does not respect the mask, so ghosting of the corners around an oval may be visible, depending on the colors involved.

To use a masked boolean gadget, fill out an instance of the **BoolInfo** structure. This structure contains a pointer to the mask plane data. The application must also set the GACT_BOOLEXTEND flag in the gadget's **Activation** field.

BOOLINFO STRUCTURE

This is the special data structure required for a masked boolean gadget. A pointer to this structure must be placed in the gadget's **SpecialInfo** field for a masked boolean gadget.

```
struct BoolInfo
    {
    UWORD  Flags;
    UWORD  *Mask;
    ULONG  Reserved;
    };
```

Flags

Flags must be given the value BOOLMASK.

Mask

This is a bit mask for highlighting and picking the gadget. Construct the mask as a single plane of Image data would be built. The image's width and height are determined by the width and height of the gadget's select box. The mask data must be in chip memory.

Reserved

Set this field to NULL.

MUTUAL EXCLUDE

Mutual exclusion of boolean gadgets (sometimes referred to as ''radio buttons'') is not directly supported by Intuition. This section describes the method an application should use to implement this feature. It is up to the application to handle the manipulation of excluded gadgets in an Intuition compatible way. The program must proceed with caution so as to maintain the synchronization of the gadget and its imagery. The rules provided in this section for the implementation of mutual exclude gadgets minimize the risk and complexity of the application. Other techniques may seem to work with simple input, but may fail in subtle ways when stressed.

Gadget Type for Mutual Exclusion

To implement mutual exclusion, gadgets must be hit-select (not GACT_TOGGLESELECT) boolean gadgets, with the GACT_IMMEDIATE activation type (never GACT_RELVERIFY). All state changes must be executed upon receiving the IDCMP_GADGETDOWN message for the gadgets. Failure to do this could introduce subtle out-of-phase imagery problems.

Gadget Highlighting for Mutual Exclusion

When using complement mode highlighting, the image supplied must be at least the size of the complemented area (the gadget select box). An extended boolean gadget with a mask may be used to constrain the area that is highlighted.

Alternate image highlighting may be used provided the two images have exactly the same size and position. Likewise, a border and alternate border may be used provided the two borders are identical in shape and position, differing only in color.

Do *not* use other combinations for mutual exclude gadgets such as a gadget with a border that uses complement mode highlighting or a gadget which uses highlighting by drawing a box. See the section on "Updating a Gadget's Imagery" for more information.

Handling of Mutually Exclusive Gadgets

Use **RemoveGList()** to remove the boolean gadget from the window or requester. Set or clear the GFLG_SELECTED flag to reflect the displayed state of the gadget. Replace the gadget using **AddGList()** and refresh its imagery with **RefreshGList()**. Of course, several gadgets may be processed with a single call to each of these functions.

Proportional Gadget Type

Proportional gadgets allow an application to get or display an amount, level, or position by moving a slidable knob within a track. They are called *proportional* gadgets because the size and position of the knob is proportional to some application-defined quantity, for example the size of a page, and how much and which part of the page is currently visible.

An example of using proportional gadgets is available in the "Intuition Windows" chapter. The SuperBitMap window example, lines.c, uses proportional gadgets to control the position of the bitmap within the window.

Proportional gadgets are made up of a *container*, which is the full size of the gadget, and a *knob*, that travels within the container. Changing the current value of the gadget is done by dragging the knob, or clicking in the container around the knob. Dragging the knob performs a smooth transition from one value to the next, while clicking in the container jumps to the next page or setting. The KNOBHIT flag in the **PropInfo** structure is available to allow the program to determine if the gadget was changed by dragging the knob or by clicking in the container. If the flag is set, the user changed the value by dragging the knob.

Proportional gadgets allow display and control of fractional settings on the vertical axis, the horizontal axis or both. While the number of settings has a theoretical limit of 65,536 positions, the actual positioning of the gadget through sliding the knob is limited by the resolution of the screen. Further control is available by clicking in the container, although this often is not convenient for the user. Button or arrow gadgets are often provided for fine tuning of the setting of the gadget.

NEW 3D LOOK PROPORTIONAL GADGETS

Set the PROPNEWLOOK flag in the **PropInfo Flags** field to get the new 3D look. The new 3D look proportional gadgets have a dithered pattern in the container and updated knob imagery. The knob dimensions are also slightly changed for those proportional gadgets with a border.

Set the PROPBORDERLESS flag in the **PropInfo Flags** field if no border around the container is desired. Setting this flag with PROPNEWLOOK will provide a 3D knob.

> *Proportional gadgets and the New 3D Look.* To create prop gadgets that have the same look as the rest of the system, set the PROPNEWLOOK flag and clear the PROPBORDERLESS flag. It is recommended that applications follow this guideline to maintain a compatible look and feel for all gadgets in the system.

New look proportional gadgets placed in the border of a window will change to an inactive display state when the window is deactivated. This only happens to gadgets that have the PROPNEWLOOK flag set and are in the window border. In the inactive state, the knob is filled with BACKGROUNDPEN.

LOGICAL TYPES OF PROPORTIONAL GADGETS

There are two usual ways in which proportional gadgets are used (corresponding to the *scroller* and *slider* gadgets of the GadTools library). The only difference between sliders and scrollers is the way they are managed internally by the application. The GadTools library provides a high level interface to proportional gadgets, simplifying the management task for these types of objects.

Scrollers

The scroller controls and represents a limited window used to display a large amount of data. For instance, a text editor may be operating on a file with hundreds of lines, but is only capable of displaying twenty or thirty lines at a time.

In a scroller, the container of the gadget is analogous to the total amount of data, while the knob represents the window. (Note that *window* here is used as an abstract concept and does not necessarily mean *Intuition window*. It just means a display area reserved for viewing the data.)

The size of the knob with respect to its container is proportional to the size of the window with respect to the total data. Thus, if the window can display half the data, the knob should be half the size of the container. When the amount of data is smaller than the window size, the knob should be as large as its container.

The position of the knob with respect to its container is also proportional to the position of the window with respect to the total data. Thus, if the knob starts half way down the container, the top of the window should display information half way into the data.

Scrollers may be one or two dimensional. One dimensional scrollers are used to control linear data; such as a text file, which can be viewed as a linear array of strings. Such scrollers only slide on a single axis.

Two dimensional scrollers are used to control two dimensional data, such as a large graphic image. Such a scroller can slide on both the horizontal and vertical axes, and the knob's horizontal and vertical size and position should be proportional to the window's size and position in the data set.

Multi-dimensional data may also be controlled by a number of one dimensional scrollers, one for each axis. The Workbench windows provide an example of this, where one scroller is used for control of the x-axis of the window and another scroller is used for control of the y-axis of the window. In this case, the size and position of the knob is proportional to the size and position of the axis represented by the gadget.

If the window has a sizing gadget and has a proportional gadget is the right or bottom border, the sizing gadget is usually placed into the border containing the proportional gadget, as the border has already been expanded to contain the gadget. If the window has proportional gadgets in both the right and the bottom borders, place the sizing gadget into both borders. This creates evenly sized borders that match the height and width of the sizing gadget, i.e. it is only done for visual effect.

Sliders

The slider is used to pick a specific value within a set. Usually the set is ordered, but this is not required. An example of this would be choosing the volume of a sound, the speed of an animation or the brightness of a color. Sliders can move on either the vertical or horizontal axis. A slider that moves on both the horizontal and the vertical axis could be created to choose two values at once.

An example slider which picks an integer between one and ten, should have the following attributes:

- It should slide on only one axis.
- Values should be evenly distributed over the length of the slider.
- Clicking in the container to either side of the knob should increase (or decrease) the value by one unit.

Stylistically, sliding the knob to the right or top should increase the value, while sliding it to the left or down should decrease the value. Note that the orientation of proportional gadgets is correct for scrollers (where the minimum value is topmost or leftmost), but is vertically inverted for sliders. Thus, well-behaved vertical sliders need to invert their value somewhere in the calculations (or else the maximum will end up at the bottom).

PROPORTIONAL GADGET COMPONENTS

A proportional gadget has several components that work together. They are the container, the knob, the pot variables and the body variables.

The Container

The container is the area in which the knob can move. It is actually the select box of the gadget. The size of the container, like that of any other gadget select box, can be relative to the size of the window. The position of the container can be relative to any of the Intuition window's border.

Clicking in the container around the knob will increment or decrement the value of the gadget (the pot variables) by the appropriate amount (specified in the body variables). The knob will move towards the point clicked when action is taken.

The Knob

The knob may be manipulated by the user to quickly change the pot variables. The knob acts like a real-world proportional control. For instance, a knob restricted to movement on a single axis can be thought of as a control such as the volume knob on a radio. A knob that moves on both axes is analogous to the control stick of an airplane.

The user can directly move the knob by dragging it on the vertical or horizontal axis. The knob may be indirectly moved by clicking within the select box around the knob. With each click, the pot variable is increased or decreased by one increment, defined by the settings of the body variables.

The current position of the knob reflects the pot value. A pot value of zero will place the knob in the top or leftmost position, a value of MAXPOT will place the knob at the bottom or rightmost position.

The application can provide its own imagery for the knob or it may use Intuition's auto-knob. The auto-knob is a rectangle that changes its width and height according to the current body settings. The auto-knob is proportional to the size of the gadget. Therefore, an auto-knob can be used in a proportional gadget whose size is relative to the size of the window, and the knob will maintain the correct size, relative to the size of the container.

> *Use Separate Imagery for Proportional Gadgets.* These **Image** structures may not be shared between proportional gadgets, each must have its own. Again, do not share the **Image** structures between proportional gadgets. This does not work, either for auto-knob or custom imagery.

> *Use Only One Image for the Knob.* Proportional gadget knob images may not be a list of images. These must be a single image, initialized and ready to display if a custom image is used for the knob.

The Pot Variables

The **HorizPot** and **VertPot** variables contain the actual proportional values entered into or displayed by the gadget. The word pot is short for potentiometer, which is an electrical analog device used to adjust a value within a continuous range.

The proportional gadget pots allow the program to set the current position of the knob within the container, or to read the knob's current location.

The pot variable is a 16-bit, unsigned variable that contains a value ranging from zero to 0xFFFF. For clarity, the constant MAXPOT is available, which is equivalent to 0xFFFF. A similar constant MAXBODY is available for the body variables. As the pot variables are only 16 bits, the resolution of the proportional gadgets has a maximum of 65,536 positions (zero to 65,535).

The values represented in the pot variables are usually translated or converted to a range of numbers more useful to the application. For instance, if a slider covered the range one to three, pot values of zero to 16,383 would represent one, values of 16,384 to 49,151 would represent two and values of 49,152 to 65,535 would represent three. The method of deriving these numbers is fairly complex, refer to the sample code below for more information.

There are two pot variables, as proportional gadgets are adjustable on the horizontal axis, the vertical axis or both. The two pot variables are independent and may be initialized to any 16-bit, unsigned value.

Pot values change while the user is manipulating the gadget. The program may read the values in the pots at any time after it has submitted the gadget to the user via Intuition. The values always have the current settings as adjusted by the user.

The Body Variables

The **HorizBody** and **VertBody** variables describe the standard increment by which the pot variables change and the relative size of the knob when auto-knob is used. The increment, or typical step value, is the value added to or subtracted from the internal knob position when the user clicks in the container around the knob. For example, a proportional gadget for color mixing might allow the user to add or subtract 1/16 of the full value each time, thus the body variable should be set to MAXBODY / 16.

Body variables are also used in conjunction with the auto-knob (described above) to display for the user how much of the total quantity of data is displayed. Additionally, the user can tell at a glance that clicking in the container around the knob will advance the position by an amount proportional to the size of the knob.

For instance, if the data is a fifteen line text file, and five lines are visible in the display, then the body variable should be set to one third of MAXBODY. In this case, the auto-knob will fill one third of the container, and clicking in the container ahead of the knob will advance the position in the file by one third.

For a slider, the body variables are usually set such that the full percentage increment is represented. This is not always so for a scroller. With a scroller, some overlap is often desired between successive steps. For example, when paging through a text editor, one or two lines are often left on screen from the previous page, making the transition easier on the user.

The two body variables may be set to the same or different increments. When the user clicks in the container, the pot variables are adjusted by an amount derived from the body variables.

Using the Body and Pot Values

The body and pot values of a proportional gadget are ''Intuition friendly'' numbers, in that they represent concepts convenient to Intuition, and not to the application. The application must translate these numbers to internal values before acting on them.

Functions for Using a Scroller

```
/*
**   FindScrollerValues( )
**
** Function to calculate the Body and Pot values of a proportional gadget
** given the three values total, displayable, and top, representing the
** total number of items in a list, the number of items displayable at one
** time, and the top item to be displayed.  For example, a file requester
** may be able to display 10 entries at a time.  The directory has 20
** entries in it, and the top one displayed is number 3 (the fourth one,
** counting from zero), then total = 20, displayable = 10, and top = 3.
**
** Note that this routine assumes that the displayable variable is greater
** than the overlap variable.
**
** A final value, overlap, is used to determine the number of lines of
** "overlap" between pages.  This is the number of lines displayed from the
** previous page when jumping to the next page.
*/
void FindScrollerValues(UWORD total, UWORD displayable, UWORD top,
                        WORD overlap, UWORD *body, UWORD *pot)
{
UWORD hidden;

/* Find the number of unseen lines: */
hidden = max(total - displayable, 0);

/* If top is so great that the remainder of the list won't even
** fill the displayable area, reduce top:
*/
if (top > hidden)
    top = hidden;

/* body is the relative size of the proportional gadget's knob.  Its size
** in the container represents the fraction of the total that is in view.
** If there are no lines hidden, then body should be full-size (MAXBODY).
** Otherwise, body should be the fraction of (the number of displayed
** lines - overlap) / (the total number of lines - overlap).  The "- overlap"
```

```
** is so that when the  user scrolls by clicking in the container of the
** scroll gadget, then there is some overlap between the two views.
*/
(*body) = (hidden > 0) ?
    (UWORD) (((ULONG) (displayable - overlap) * MAXBODY) / (total - overlap)) :
    MAXBODY;

/* pot is the position of the proportional gadget knob, with zero meaning that
** the scroll gadget is all the way up (or left), and full (MAXPOT) meaning
** that the scroll gadget is all the way down (or right).  If we can see all
** the lines, pot should be zero.  Otherwise, pot is the top displayed line
** divided by the number of unseen lines.
*/
(*pot) = (hidden > 0) ? (UWORD) (((ULONG) top * MAXPOT) / hidden) : 0;
}


/*
**   FindScrollerTop( )
**
** Function to calculate the top line that is displayed in a proportional
** gadget, given the total number of items in the list and the number
** displayable, as well as the HorizPot or VertPot value.
*/
UWORD FindScrollerTop(UWORD total, UWORD displayable, UWORD pot)
{
UWORD top, hidden;

/* Find the number of unseen lines: */
hidden = max(total - displayable, 0);

/* pot can be thought of as the fraction of the hidden lines that are before
** the displayed part of the list, in other words a pot of zero means all
** hidden lines are after the displayed part of the list (i.e. top = 0),
** and a pot of MAXPOT means all the hidden lines are before the displayed
** part (i.e. top = hidden).
**
** MAXPOT/2 is added to round up values more than half way to the next position.
*/
top = (((ULONG) hidden * pot) + (MAXPOT/2)) >> 16;

/* Once you get back the new value of top, only redraw your list if top
** changed from its previous value.  The proportional gadget may not have
** moved far enough to change the value of top.
*/
return(top);
}
```

Functions for Using a Slider

```
/*
**     FindSliderValues( )
**
** Function to calculate the Body and Pot values of a slider gadget given the
** two values numlevels and level, representing the number of levels available
** in the slider, and the current level.  For example, a Red, Green, or Blue
** slider would have (currently) numlevels = 16, level = the color level (0-15).
*/
void FindSliderValues(UWORD numlevels, UWORD level, UWORD *body, UWORD *pot)
{
/* body is the relative size of the proportional gadget's body.
** Clearly, this proportion should be 1 / numlevels.
*/

if (numlevels > 0)
    (*body) = (MAXBODY) / numlevels;
else
    (*body) = MAXBODY;

/* pot is the position of the proportional gadget body, with zero meaning that
** the slider is all the way up (or left), and full (MAXPOT) meaning that the
** slider is all the way down (or right).
**
** For slider gadgets the derivation is a bit ugly:
```

```
**
** We illustrate a slider of four levels (0, 1, 2, 3) with the slider at
** level 2.  The key observation is that pot refers the the leading edge of
** the knob, and as such MAXPOT is not all the way to the right, but is one
** body's width left of that.
**
** Level:   0       1       2       3
**       ---------------------------------
**       |       |       |*******|       |
**       |       |       |*******|       |
**       |       |       |*******|       |
**       |       |       |*******|       |
**       ---------------------------------
**       |               |       |
**       0              pot    MAXPOT
**
** From which we observe that pot = MAXPOT * (level/(numlevels-1))
*/

if (numlevels > 1)
    {
    (*pot) = (((ULONG)MAXPOT) * level)/(numlevels-1);
    }
else
    {
    (*pot) = 0;
    }
}


/*
**     FindSliderLevel( )
**
** Function to calculate the level of a slider gadget given the total number
** of levels as well as the HorizPot or VertPot value.
*/
UWORD FindSliderLevel(UWORD numlevels, UWORD pot)
{
UWORD level;

/* We illustrate a 4-level slider (0, 1, 2, 3) with the knob on the transition
** point between calling it at levels 1 and 2.
**
** Level:   0       1       2       3
**       ---------------------------------
**       |       |   ***|***     |       |
**       |       |   ***|***     |       |
**       |       |   ***|***     |       |
**       |       |   ***|***     |       |
**       ---------------------------------
**       |           |           |
**       0          pot        MAXPOT
**
**
** We've already shown that the vertical lines (which represent the natural
** position of the knob for a given level are:
**
**     pot = MAXPOT * (level/(numlevels-1))
**
** and we see that the threshold between level and level-1 is half-way between
** pot(level) and pot(level-1), from which we get
**
**     level = (numlevels-1) * (pot/MAXPOT) + 1/2
*/

if (numlevels > 1)
    {
    level = (((ULONG)pot) * (numlevels-1) + MAXPOT/2) / MAXPOT;
    }
else
    {
    level = 0;
    }

return(level);
}
```

INITIALIZATION OF A PROPORTIONAL GADGET

The proportional gadget is initialized like any other gadget, with the addition of the **PropInfo** structure.

Initialization of the PropInfo Structure

This is the special data required by the proportional gadget.

```
struct PropInfo
    {
    UWORD Flags;
    UWORD HorizPot;
    UWORD VertPot;
    UWORD HorizBody;
    UWORD VertBody;
    UWORD CWidth;
    UWORD CHeight;
    UWORD HPotRes, VPotRes;
    UWORD LeftBorder;
    UWORD TopBorder;
    };
```

Flags

In the **Flags** variable, the following flag bits are of interest:

PROPBORDERLESS

Set the PROPBORDERLESS flag to create a proportional gadget without a border.

AUTOKNOB

Set the AUTOKNOB flag in the **Flags** field to use the auto-knob, otherwise the application must provide knob imagery.

FREEHORIZ and **FREEVERT**

Set the FREEHORIZ flag to create a gadget that adjust left-to-right, set the FREEVERT flag for top-to-bottom movement. Both flags may be set in a single gadget.

PROPNEWLOOK

Set the PROPNEWLOOK flag to create a gadget with the new look. If this flag is not set, the gadget will be rendered using a V34 compatible design.

KNOBHIT

The KNOBHIT flag is set by Intuition when this knob is hit by the user.

HorizPot and **VertPot**

Initialize the **HorizPot** and **VertPot** variables to their starting values before the gadget is added to the system. The variables may be read by the application. The gadget must be removed before writing to these variables, or they may be modified with **NewModifyProp()**.

HorizBody and **VertBody**

Set the **HorizBody** and **VertBody** variables to the desired increment. If there is no data to show or the total amount displayed is less than the area in which to display it, set the body variables to the maximum, MAXBODY.

The remaining variables and flags are reserved for use by Intuition.

Initialization of the Gadget Structure

In the **Gadget** structure, set the **GadgetType** field to GTYP_PROPGADGET and place the address of the **PropInfo** structure in the **SpecialInfo** field.

When using AUTOKNOB, the **GadgetRender** field must point to an **Image** structure. The **Image** need not be initialized when using AUTOKNOB, but the structure must be provided. These **Image** structures may not be shared between gadgets, each must have its own.

To use application imagery for the knob, set **GadgetRender** to point to an initialized **Image** structure. If the knob highlighting is done by alternate image (GFLG_GADGHIMAGE), the alternate image must be the same size and type as the normal knob image.

MODIFYING AN EXISTING PROPORTIONAL GADGET

To change the flags and the pot and body variables after the gadget is displayed, the program can call **NewModifyProp()**.

```
void NewModifyProp( struct Gadget *gadget, struct Window *window, struct Requester *requester,
       unsigned long flags, unsigned long horizPot, unsigned long vertPot,
       unsigned long horizBody, unsigned long vertBody, long numGad );
```

The gadget's internal state will be recalculated and the imagery will be redisplayed to show the new state. When **numGads** (in the prototype above) is set to all ones, **NewModifyProp()** will only update those parts of the imagery that have changed, which is much faster than removing the gadget, changing values, adding the gadget back and refreshing its imagery.

String Gadget Type

A string gadget is an area of the display in which a single field of character data may be entered. When a string gadget is activated, either by the user or by the application, a cursor appears prompting the user to enter some text. Any characters typed will be placed into the active string gadget, unless the gadget is deactivated by other mouse activity or program interaction.

In Release 2, the system also supports tabbing between a group of string gadgets. In this mode, pressing the tab key will advance the active gadget to the next string gadget and pressing shifted tab will advance to the previous string gadget.

Control characters are generally filtered out, but may be entered by pressing the Left Amiga key with the desired control character. The filtering may be disabled by the program, or by the user via the IControl Preferences editor.

String gadgets feature auto-insert, which allows the user to insert characters wherever the cursor is. Overwrite mode is also available, and the application may toggle the gadget between the two modes.

When the user activates a string gadget with the mouse, the gadget's cursor moves to the position of the mouse. The user may change the position of the cursor both with the cursor keys and with the mouse pointer.

A number of simple, keyboard driven editing functions are available to the user. These editing functions are shown in the following table.

Table 5-1: Editing Keys and Their Functions

Key	Function
←	Cursor to previous character.
Shift ←	Cursor to beginning of string.
→	Cursor to next character.
Shift →	Cursor to end of string.
Del	Delete the character under the cursor. Does nothing in fixed field mode.
Shift Del	Delete from the character under the cursor to the end of the line. Does nothing in fixed field mode.
Backspace	Delete the character to left of cursor. In fixed field mode, move cursor to previous character.
Shift Backspace	Delete from the character to the left of the cursor to the start of the line. In fixed field mode, move cursor to beginning of string.
Return or Enter	Terminate input and deactivate the gadget. If the GACT_RELVERIFY activation flag is set, the program will receive a IDCMP_GADGETUP event for this gadget.
Right Amiga Q	Undo (cancel) the last editing change to the string.
Right Amiga X	Clears the input buffer. The undo buffer is left undisturbed. In fixed field mode, move cursor to beginning of string.

The following additional editing functions are available only when ''Filter Control Characters'' is on for the string gadget. Control character filtering is only available if the IControl preferences editor has ''Text Gadget Filter'' selected and the individual gadget does not have SGM_NOFILTER set.

Table 5-2: Additional Editing Keys and Their Functions

Key	Function
Ctrl A	Jump cursor to start of buffer.
Ctrl H	Delete the character to the left of the cursor. In fixed field mode, move cursor to previous character.
Ctrl K	Delete from the character under the cursor to the end of the string. Does nothing in fixed field mode.
Ctrl M	Equivalent to Return or Enter (end gadget).
Ctrl W	Delete the previous word. In fixed field mode, jump cursor to the start of the previous word.
Ctrl U	Delete from the character to the left of the cursor to the start of the buffer. In fixed field mode, jump cursor to the start of the buffer.
Ctrl X	Clears the input buffer (like Right Amiga X). In fixed field mode, jump cursor to the start of the buffer.
Ctrl Z	Jump cursor to end of buffer.

INTEGER GADGET TYPE

The integer gadget is really a special case of the string gadget type. Initialize the gadget as a string gadget, then set the GACT_LONGINT flag in the gadget's **Activation** field.

The user interacts with an integer gadget using exactly the same rules as for a string gadget, but Intuition filters the input, allows the user to enter only a plus or minus sign and digits. The integer gadget returns a signed 32-bit integer in the **StringInfo** variable **LongInt**.

To initialize an integer gadget to a value, preload the input buffer with an ASCII representation of the initial integer. It is not sufficient to initialize the gadget by merely setting a value in the **LongInt** variable.

Integer gadgets have the **LongInt** value updated whenever the ASCII contents of the gadget changes, and again when the gadget is deactivated.

STRING GADGET IDCMP MESSAGES

If the application has specified the GACT_RELVERIFY activation flag, it will be sent an IDCMP_GADGETUP message when the gadget is properly deactivated. This happens when Return or Enter is pressed, when tabbing to the next string gadget (where supported), and when a custom string editing hook returns SGA_END.

The gadget may become inactive without the application receiving an IDCMP_GADGETUP message. This will happen if the user performs some other operation with the mouse or if another window is activated. The gadget may still contain updated, valid information even though the IDCMP_GADGETUP message was not received.

PROGRAM CONTROL OF STRING GADGETS

ActivateGadget() allows the program to activate a string gadget (and certain custom gadgets). If successful, this function has the same effect as the user clicking the mouse select button when the mouse pointer is within the gadget's select box and any subsequent keystrokes will effect the gadget's string.

```
BOOL ActivateGadget( struct Gadget *gadget, struct Window *window, struct Requester *requester );
```

This function will fail if the user is in the middle of some other interaction, such as menu or proportional gadget operation. In that case it returns FALSE, otherwise it returns TRUE. The window or requester containing the string gadget to be activated must itself be open and active. Since some operations in Intuition may occur after the function that initiates them completes, calling **ActivateGadget()** after **OpenWindowTagList()** or **Request()** is no guarantee that the gadget will actually activate. Instead, call **ActivateGadget()** only after having received an IDCMP_ACTIVEWINDOW or IDCMP_REQSET message for a newly opened window or requester, respectively.

> *The Window Active Message Is Required.* It is incorrect to simply insert a small delay between the call to **OpenWindowTagList()** or **Request()** and the call to **ActivateGadget()**. Such schemes fail under various conditions, including changes in processor speed and CPU loading.

If you want to activate a string gadget in a newly opened window that has a shared IDCMP **UserPort**, there is an additional complication. Sharing **UserPorts** means that the window is opened without any IDCMP messages enabled, and only later is **ModifyIDCMP()** called to turn on message passing. If the newly opened window becomes active before **ModifyIDCMP()** is called, the IDCMP_ACTIVEWINDOW message will not be received (because IDCMP message passing was off at the time). The following code will handle this problem:

```
BOOL activated;

/* Open window with NULL IDCMPFlags */
win = OpenWindow( ... );

/* Set the UserPort to your shared port, and turn on message passing,
 * which includes the IDCMP_ACTIVEWINDOW message.
 */
win->UserPort = sharedport;
ModifyIDCMP( win, ... | IDCMP_ACTIVEWINDOW | ... );

/* If the window became active before the ModifyIDCMP() got executed,
 * then this ActivateGadget() can succeed.  If not, then this
 * ActivateGadget() might be too early, but in that case, we know
 * we'll receive the IDCMP_ACTIVEWINDOW event.  We handle that below.
 */
 activated = ActivateGadget( stringgad, win, NULL );
```

and later, in the event loop:

```
if ( (msg->Class == ACTIVEWINDOW) && ( !activated ) )
    success = ActivateGadget(stringgad,...);
```

Note however that a window which has the WA_Activate attribute is not guaranteed to be activated upon opening. Certain conditions (like an active string gadget in another window) will prevent the automatic initial activation of the window. Therefore, do not let your code depend on receiving the initial IDCMP_ACTIVEWINDOW message.

String Gadget Example

The values of a string gadget may be updated by removing the gadget, modifying the information in the **StringInfo** structure, adding the gadget back and refreshing its imagery.

```
;/* updatestrgad.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 updatestrgad.c
Blink FROM LIB:c.o,updatestrgad.o TO updatestrgad LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**
** updatestrgad.c - Show the use of a string gadget.  Shows both the use of
** ActivateGadget() and how to properly modify the contents of a string gadget.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <string.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
/* our function prototypes */
VOID updateStrGad(struct Window *win, struct Gadget *gad, UBYTE *newstr);
VOID handleWindow(struct Window *win, struct Gadget *gad);

struct Library *IntuitionBase;

/* NOTE that the use of constant size and positioning values are
** not recommended; it just makes it easy to show what is going on.
** The position of the gadget should be dynamically adjusted depending
** on the height of the font in the title bar of the window.  This
** example adapts the gadget height to the screen font. Alternately,
** you could specify your font under V37 with the StringExtend structure.
*/
#define BUFSIZE (100)
#define MYSTRGADWIDTH (200)
#define MYSTRGADHEIGHT (8)

UWORD strBorderData[] =
    {
    0,0, MYSTRGADWIDTH + 3,0, MYSTRGADWIDTH + 3,MYSTRGADHEIGHT + 3,
    0,MYSTRGADHEIGHT + 3, 0,0,
    };
struct Border strBorder =
    {
    -2,-2,1,0,JAM1,5,strBorderData,NULL,
    };
UBYTE strBuffer[BUFSIZE];
UBYTE strUndoBuffer[BUFSIZE];
struct StringInfo strInfo =
    {
    strBuffer,strUndoBuffer,0,BUFSIZE, /* compiler sets remaining fields to zero */
    };
struct Gadget strGad =
    {
    NULL, 20,20,MYSTRGADWIDTH,MYSTRGADHEIGHT,
    GFLG_GADGHCOMP, GACT_RELVERIFY | GACT_STRINGCENTER,
    GTYP_STRGADGET, &strBorder, NULL, NULL,0,&strInfo,0,NULL,
    };

#define ANSCNT 4
UBYTE *answers[ANSCNT] = {"Try again","Sorry","Perhaps","A Winner"};
int ansnum = 0;
UBYTE *activated_txt = "Activated";

/*   main - show the use of a string gadget.
*/
VOID main(int argc, char **argv)
{
struct Window *win;

/* make sure to get intuition version 37, for OpenWindowTags() */
IntuitionBase = OpenLibrary("intuition.library", 37);
if (IntuitionBase)
    {
    /* Load a value into the string gadget buffer.
    ** This will be displayed when the gadget is first created.
    */
    strcpy(strBuffer, "START");

    if (win = OpenWindowTags(NULL,
                            WA_Width, 400,
                            WA_Height, 100,
                            WA_Title,"Activate Window, Enter Text",
                            WA_Gadgets, &strGad,
                            WA_CloseGadget, TRUE,
                            WA_IDCMP, IDCMP_ACTIVEWINDOW |
                                IDCMP_CLOSEWINDOW | IDCMP_GADGETUP,
                            TAG_END))
        {
        handleWindow(win,&strGad);

        CloseWindow(win);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

```
/*
** Process messages received by the window.  Quit when the close gadget
** is selected, activate the gadget when the window becomes active.
*/
VOID handleWindow(struct Window *win, struct Gadget *gad)
{
struct IntuiMessage *msg;
struct Gadget *gadget;
ULONG  class;

for (;;)
    {
    Wait(1L << win->UserPort->mp_SigBit);
    while (msg = (struct IntuiMessage *)GetMsg(win->UserPort))
        {
        /* Stash message contents and reply, important when message
        ** triggers some lengthy processing
        */
        class = msg->Class;
      /* If it's a gadget message, IAddress points to Gadget */
        if((class == IDCMP_GADGETUP)||(class == IDCMP_GADGETDOWN))
                gadget = (struct Gadget *)msg->IAddress;
        ReplyMsg((struct Message *)msg);

        switch (class)
            {
            case IDCMP_ACTIVEWINDOW:
                /* activate the string gadget.  This is how to activate a
                ** string gadget in a new window--wait for the window to
                ** become active by waiting for the IDCMP_ACTIVEWINDOW
                ** event, then activate the gadget.  Here we report on
                ** the success or failure.
                */
                if(ActivateGadget(gad,win,NULL))
                    updateStrGad(win,gad,activated_txt);
                break;
            case IDCMP_CLOSEWINDOW:
                /* here is the way out of the loop and the routine.
                ** be sure that the message was replied...
                */
                return;
                break;
            case IDCMP_GADGETUP:
                /* If user hit RETURN in our string gadget for demonstration,
                ** we will change what he entered.  We only have 1 gadget,
                ** so we don't have to check which gadget.
                */
                updateStrGad(win, &strGad, answers[ansnum]);
            if(++ansnum > ANSCNT) ansnum = 0;  /* point to next answer */
                break;
            }
        }
    }
}



/*
** Routine to update the value in the string gadget's buffer, then
** activate the gadget.
*/
VOID updateStrGad(struct Window *win, struct Gadget *gad, UBYTE *newstr)
{
/* first, remove the gadget from the window.  this must be done before
** modifying any part of the gadget!!!
*/
RemoveGList(win,gad,1);

/* For fun, change the value in the buffer, as well as the cursor and
** initial display position.
*/
strcpy(((struct StringInfo *)(gad->SpecialInfo))->Buffer, newstr);
((struct StringInfo *)(gad->SpecialInfo))->BufferPos = 0;
((struct StringInfo *)(gad->SpecialInfo))->DispPos   = 0;

/* Add the gadget back, placing it at the end of the list (~0)
```

```
** and refresh its imagery.
*/
AddGList(win,gad,~0,1,NULL);
RefreshGList(gad,win,NULL,1);

/* Activate the string gadget */
ActivateGadget(gad,win,NULL);
}
```

TABBING BETWEEN STRING GADGETS

The Amiga allows tabbing to the next string gadget in a window or requester and shifted tabbing to the previous string gadget. This function operates starting with V37.

If the GFLG_TABCYCLE flag is set, this string participates in cycling activation with Tab or Shift Tab. If only a single gadget has this flag set, then the Tab keys will have no effect. If one of the Tab keys is pressed while in a string gadget without GFLG_TABCYCLE set, nothing will happen, even though other string gadgets may have the flag set.

Activation order is determined by the order of the string gadgets in the gadget list, following the **NextGadget** link. The tab key will advance to the next string gadget with GFLG_TABCYCLE set, shifted tab will move to the previous gadget. To order gadgets for tabbing (next/ previous string gadget), place them in the correct order in the gadget list when they are added to the system. This order must be maintained if the gadgets are removed and put back, or the tabbing order will change.

The tab keys will de-activate the current gadget as if one of the Return or Enter keys had been pressed, sending an IDCMP_GADGETUP message to the application. The application can recognize that tab was pressed by looking for 0x09 (the ASCII tab character) in the **Code** field of the IDCMP_GADGETUP **IntuiMessage**. If necessary, it can then inspect the qualifier field of that message to see if the shift key was pressed. The next string gadget with GFLG_TABCYCLE set will be activated, with shifted tab activating the previous string gadget.

GADGET STRUCTURE FOR STRING GADGETS

To an application, a string gadget consists of a standard **Gadget** structure along with an entry buffer, an undo buffer and a number of extensions.

For a string gadget, set the **GadgetType** field in the **Gadget** structure to GTYP_STRGADGET. Set the **SpecialInfo** field to point to an instance of a **StringInfo** structure, which must be initialized by the application.

The container for a string gadget is its select box. The application specifies the size of the container. As the user types into the string gadget, the characters appear in the gadget's container.

String gadgets may hold more characters than are displayable in the container. To use this feature, the application simply provides a buffer that is larger than the number of characters that will fit in the container. This allows the user to enter and edit strings that are much longer than the visible portion of the buffer. Intuition maintains the cursor position and scrolls the text in the container as needed.

The application may specify the justification of the string in the container. The default is GACT_STRINGLEFT, or left justification. If the flag GACT_STRINGCENTER is set, the text is center justified; if GACT_STRINGRIGHT is set, the text is right justified.

When the gadget is activated, the select box contents are redrawn, including the background area. If GFLG_STRINGEXTEND is set for the gadget or the gadget is using a proportional font by default, then the entire select box will be cleared regardless of the font size or **StringInfo.MaxChars** value. For compatibility reasons, if the string gadget is not extended then the following conditions apply (see the section on ''Extending String Gadgets'' for more information).

- If the font is monospace (not proportional), the width of the gadget will be rounded down to an even multiple of the font width.

- If the string gadget is left justified (GACT_STRINGLEFT), a maximum of **StringInfo.MaxChars** times the font width pixels of space will be cleared. Thus, if **MaxChars** is 3 (two characters plus the trailing NULL) and the font width is 8, then a maximum of 3 * 8 = 24 pixels will be cleared. If the font defaults to a proportional font, then the width returned by **FontExtent()** will be used as the character width.

No facilities are provided to place imagery within the select box of a string gadget.

String Gadget Imagery and Highlighting

Any type of image may be supplied for the rendering of a string gadget--image, border, or no image at all. The highlighting for a string gadget must be the complementing type (GFLG_GADGHCOMP). Alternate imagery may not be used for highlighting.

STRINGINFO STRUCTURE

String gadgets require their own special structure called the **StringInfo** structure.

```
struct StringInfo
    {
    UBYTE *Buffer;
    UBYTE *UndoBuffer;
    WORD BufferPos;
    WORD MaxChars;
    WORD DispPos;
    WORD UndoPos;
    WORD NumChars;
    WORD DispCount;
    WORD CLeft, CTop;
    struct StringExtend *Extension;
    LONG LongInt;
    struct KeyMap *AltKeyMap;
    };
```

Buffer

The application must supply an input buffer (**Buffer**) and an optional undo buffer (**UndoBuffer**) for the gadget. The input buffer is where data typed into the gadget is placed by Intuition. The program can examine this buffer at any time.

A string copied into the input buffer before the gadget is added to the system will be displayed in the gadget when it is displayed, and may then be edited by the user. The input buffer may be initialized to any starting value, as long as the initial string is NULL terminated and fits within the buffer. To initialize the buffer to the empty string (no characters), put a NULL in the first position of the buffer.

Integer gadgets must have the ASCII value of the initial number placed into the **Buffer** before the gadget is added to the system.

UndoBuffer

If a string gadget has an undo buffer, the undo feature will be enabled. "Undo" allows the user to revert to the initial string (the value in the buffer before gadget activation) at any time before the gadget becomes inactive. The **UndoBuffer** is used to hold a copy of the previous string while the user edits the current string. When the gadget is activated, the **Buffer** is copied to the **UndoBuffer**. The **Buffer** may be restored at any time up to the time the gadget is deactivated, by typing right-Amiga Q.

Multiple string gadgets may share the same undo buffer as long as the buffer is as large as the largest input buffer.

MaxChars

MaxChars tells Intuition the size of the input buffer. This count includes the trailing NULL of any data entered into the buffer, so the number of characters the gadget may hold is **MaxChars** - 1.

BufferPos

BufferPos is initialized to the current position of the cursor in the buffer. **BufferPos** runs from zero to one less than the length of the string. If this position is not within the characters that will be displayed, Intuition will adjust **DispPos** for the gadget to make the cursor visible.

DispPos

DispPos is initialized to the starting character in the string to display on screen. This allows strings longer than the number of displayable characters to be positioned within the gadget. Intuition will not position the string such that there is empty character space to the right of the string and characters scrolled out of the gadget box to the left.

UndoPos, NumChars, DispCount, CLeft and CTop

These variables are maintained by Intuition and should not be modified by the application. **UndoPos** specifies the character position in the undo buffer. **NumChars** specifies the number of characters currently in the buffer. **DispCount** specifies the number of whole characters visible in the container.

Extension

The **StringInfo Extension** allows for additional control over string gadget behavior and appearance. See below for details.

LongInt

LongInt contains the integer value entered into an Integer type of string gadget. After the user has finished entering an integer, the application can read the value in this variable.

Gadget Key Mapping

By default, screen characters appear using simple ASCII key translations. If desired, the application can set up alternate key mapping. A pointer to the **KeyMap** structure is placed into the **AltKeyMap** field of the **StringInfo** structure. The GACT_ALTKEYMAP bit in the **Activation** flags of the gadget must also be set.

See the "Console Device" chapter in the *Amiga ROM Kernel Reference Manual: Devices*, and the "Keymap Library" chapter in this manual for more information about the console device and key mapping.

EXTENDED STRING GADGETS

The **StringInfo** structure may be extended by setting the GFLG_STRINGEXTEND gadget flag and placing a pointer to a **StringExtend** structure in the **StringInfo Extension** variable. GFLG_STRINGEXTEND is available beginning with V37, under V36 the application must use GACT_STRINGEXTEND to get the same functionality. Note that GACT_STRINGEXTEND is not ignored prior to V36 and should only be set in V36 or later systems. GFLG_STRINGEXTEND is ignored prior to V37.

```
struct StringExtend
    {
    struct TextFont *Font;
    UBYTE       Pens[2];
    UBYTE       ActivePens[2];
    ULONG       InitialModes;
    struct Hook *EditHook;
    UBYTE       *WorkBuffer;
    ULONG       Reserved[4];
    };
```

Font

If a font is specified in the **StringExtend** structure, that font will be used by the gadget. By default, the string gadget inherits the font of the screen on which it appears. Note that this is a pointer to an open font and not a pointer to a **TextAttr** structure.

Proportional fonts are supported in string gadgets starting with Release 2. If the select box of the gadget is not tall enough to render the font, Intuition will fall back to topaz 8.

Pens

Pens specify the pens used to render the text while the gadget is inactive. **Pens[0]** is the foreground (text) pen, **Pens[1]** is the background pen.

ActivePens

ActivePens specify the pens used to render the text while the gadget is active. **ActivePens[0]** is the foreground (text) pen, **ActivePens[1]** is the background pen.

InitialModes

These modes may be used in **StringExtend** structure **InitialModes** field.

SGM_REPLACE

If this flag is set, the string gadget will be in replace or overwrite mode. If this flag is cleared, the string gadget will be in insert mode. In replace mode, characters entered overwrite the existing characters. In insert mode, characters entered are inserted into the buffer and the following characters are advanced by one position until the buffer is full. If the buffer is full in insert mode then characters may not be entered until some are deleted.

When using this flag, always initialize **StringInfo** with an in-range value of **BufferPos**. While most changes to gadgets require the application to first remove the gadget before modifying the gadget, this flag may be toggled without removing the gadget from the gadget list. The change will take effect on the next character typed by the user.

SGM_NOFILTER

Don't filter control chars, enter them into the gadget as typed. In this mode the control character command keys for string gadgets are not active. If the user disables control character filtering from the IControl Preferences editor, there is no way for the application to turn it on for an individual string gadget. In filter mode, control characters may be entered into the string by holding the left Amiga key while the character is entered.

While most changes to gadgets require the application to first remove the gadget before modifying the gadget, this flag may be toggled without removing the gadget from the gadget list. The change will take effect on the next character typed by the user.

SGM_FIXEDFIELD

Fixed length buffer used for editing, the user cannot shorten or lengthen the string through edit operations. The field length is taken from the length of the character string in the buffer when the gadget is added to the system. Fixed field mode modifies the meanings of many of the string editing keys, as explained in the tables above. Always set SGM_REPLACE when using a fixed length buffer.

SGM_EXITHELP

Allows the help key to be heard by the application from within string gadgets. The gadget will exit immediately when the help key is pressed with the **IntuiMessage.Code** set to 0x5F (new for V37).

EditHook and WorkBuffer

EditHook and **WorkBuffer** are used for custom string editing, which is discussed below.

CUSTOM STRING EDITING

The application may choose to control the editing features provided in string gadgets used within the application. To locally install the custom string editing features, the application provides a *hook* in the **StringExtend** structure **EditHook** field.

A *hook* is a well defined calling interface for a user provided subroutine or function. Hooks are more fully described in the "Utility Library" chapter. A string gadget hook is called in the standard way, where the hook object is a pointer to a **SGWork** structure, and the hook message is a pointer to a command block. However, unlike a function callback hook, a string gadget editing hook is called on Intuition's task context, not on the application's own context. Therefore, a string gadget editing hook must not use dos.library, and may not **Wait()** on application signals or message ports, and may not call any Intuition function which might wait for Intuition.

The command block starts with either (longword) SGH_KEY or SGH_CLICK. There may be new commands added in the future, so the application should not assume that these are the only possible commands. The hook should return zero if it doesn't understand the command and non-zero if the command is supported.

The **SGWork** structure, defined in <intuition/sghooks.h>, is listed on the next page. Use this structure as the hook object for custom string editing hooks.

SGWork Structure

```
struct SGWork
    {
    struct Gadget       *Gadget;
    struct StringInfo   *StringInfo;
    UBYTE               *WorkBuffer;
    UBYTE               *PrevBuffer;
    ULONG               Modes;
    struct InputEvent   *IEvent;
    UWORD               Code;
    WORD                BufferPos;
    WORD                NumChars;
    ULONG               Actions;
    LONG                LongInt;
    struct GadgetInfo   *GadgetInfo;
    UWORD               EditOp;
    };
```

The local (application) hook may only change the **Code**, **Actions**, **WorkBuffer**, **NumChars**, **BufferPos** and **LongInt** fields. None of the other fields in the **SGWork** structure may be modified.

Gadget and StringInfo

The values in the string gadget before any modification are available through the **Gadget** and **StringInfo** pointers.

PrevBuffer

The **PrevBuffer** provides a shortcut to the old, unmodified string buffer.

WorkBuffer, BufferPos, NumChars and LongInt

WorkBuffer, **BufferPos**, **NumChars** and **LongInt** contain the values that the string gadget will take if the edits are accepted. If the edit hook updates these values, the gadget will take on the updated values.

IEvent

IEvent contains the input event that caused this call to the hook. This input event is not keymapped. Only use this event for action keys, like the Return key, function keys or the Esc key.

Code

If the input event maps to a single character, the keymapped value will be in the **Code** field. The **Code** field may also be modified, and the value placed in it will be passed back to the application in the IDCMP_GADGETUP message when SGA_END is specified in the Actions field.

GadgetInfo

A structure of information defined in <intuition/cghooks.h>. This structure is read only. See the "BOOPSI" chapter for more information.

Modes

The modes of the gadget such as insert mode, defined below.

Actions

The action taken by the edit hook, defined below.

EditOp

The type of edit operation done by the global hook, defined below.

EditOp Definitions

These values indicate the basic type of operation the global editing hook has performed on the string before the application gadget's custom editing hook gets called. Only global editing hooks must update the value in the **EditOp** field before they return. The value placed in the field should reflect the action taken.

EditOp	Action Taken by Global Hook
EO_NOOP	Did nothing.
EO_DELBACKWARD	Deleted some chars (possibly 0).
EO_DELFORWARD	Deleted some characters under and in front of the cursor.
EO_MOVECURSOR	Moved the cursor.
EO_ENTER	Enter or Return key, terminate.
EO_RESET	Current Intuition-style undo.
EO_REPLACECHAR	Replaced one character and (maybe) advanced cursor.
EO_INSERTCHAR	Inserted one character into string or added one at end.
EO_BADFORMAT	Didn't like the text data, e.g., alpha characters in a GACT_LONGINT type.
EO_BIGCHANGE	Complete or major change to the text, e.g. new string.
EO_UNDO	Some other style of undo.
EO_CLEAR	Clear the string.
EO_SPECIAL	An operation that doesn't fit into the categories here.

Actions Definitions

These are the actions to be taken by Intuition after the hook returns. Set or clear these bits in **SGWork** structure **Actions** field. A number of these flags may already be set when the hook is called.

Actions Flag	Purpose
SGA_USE	If set, use contents of **SGWork**.
SGA_END	Terminate gadget, Code field is sent to application in IDCMP_GADGETUP event code field.
SGA_BEEP	Beep (i.e., flash) the screen.
SGA_REUSE	Reuse the input event. Only valid with SGA_END.
SGA_REDISPLAY	Gadget visuals have changed, update on screen.
SGA_NEXTACTIVE	Make next possible gadget active (new for V37).
SGA_PREVACTIVE	Make previous possible gadget active (new for V37).

The SGH_KEY Command

The SGH_KEY command indicates that the user has pressed a key while the gadget is active. This may be any key, including non-character keys such as Shift, Ctrl and Alt. Repeat keys (one call per repeat) and the Amiga keys also cause the hook to be called with the SGH_KEY command. The hook is not called for "key up" events.

The SGH_KEY command must be supported by any custom string editing hook. There are no parameters following the SGH_KEY command longword. All information on the event must be derived from the **SGWork** structure.

Intuition has already processed the event and filled-in the **SGWork** structure before calling the hook. The information included in this structure includes the type of action taken (**EditOp**), the new cursor position (**BufferPos**), the new value in the buffer (**WorkBuffer**), the previous value in the buffer (**PrevBuffer**), the input event that caused this call (**IEvent**) and more.

Actions with SGH_KEY

If SGA_USE is set in the **SGWork** structure **Actions** field when the hook returns, Intuition will use the values in the **SGWork** fields **WorkBuffer**, **NumChars**, **BufferPos**, and **LongInt**; copying the **WorkBuffer** to the **StringInfo Buffer**. SGA_USE is set by Intuition prior to calling the hook, and must be cleared by the hook if the changes are to be ignored. If SGA_USE is cleared when the hook returns, the string gadget will be unchanged.

If SGA_END is set when the hook returns, Intuition will deactivate the string gadget. In this case, Intuition will place the value found in **SGWork** structure **Code** field into the **IntuiMessage.Code** field of the IDCMP_GADGETUP message it sends to the application.

If SGA_REUSE and SGA_END are set when the hook returns, Intuition will reuse the input event after it deactivates the gadget.

Starting in V37, the hook may set SGA_PREVACTIVE or SGA_NEXTACTIVE with SGA_END. This tells Intuition to activate the next or previous gadget that has the GFLG_TABCYCLE flag set.

If SGA_BEEP is set when the hook returns, Intuition will call **DisplayBeep()**. Use this if the user has typed in error, or buffer is full.

Set SGA_REDISPLAY if the changes to the gadget warrant a gadget redisplay. Changes to the cursor position require redisplay.

The SGH_CLICK Command

The SGH_CLICK command indicates that the user has clicked the select button of the mouse within the gadget select box. There are no parameters following the SGH_CLICK command longword.

Intuition will have already calculated the mouse position character cell and placed that value in **SGWork.BufferPos**. The previous **BufferPos** value remains in the **SGWork.StringInfo.BufferPos**.

Intuition will again use the **SGWork** fields listed above for SGH_KEY. That is, the **WorkBuffer**, **NumChars**, **BufferPos** and **LongInt** fields values may be modified by the hook and are used by Intuition if SGA_USE is set when the hook returns.

Actions with SGH_CLICK

SGA_END or SGA_REUSE may *not* be set for the SGH_CLICK command. Intuition will not allow gadgets which go inactive when chosen by the user. The gadget always consumes mouse events in its select box.

With SGH_CLICK, always leave the SGA_REDISPLAY flag set, since Intuition uses this when activating a string gadget.

Example String Gadget Editing Hook

```
;/* strhooks.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 strhooks.c
Blink FROM LIB:c.o,strhooks.o TO strhooks LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
**   strhooks.c - string gadget hooks demo
**
** WARNING: This file contains "callback" functions.
** You must disable stack checking (SAS -v flag) for them to work.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <utility/hooks.h>
#include <devices/inputevent.h>
#include <intuition/intuition.h>
#include <intuition/sghooks.h>
#include <graphics/displayinfo.h>

#include <clib/intuition_protos.h>
#include <clib/utility_protos.h>
#include <clib/exec_protos.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* our function prototypes */
BOOL IsHexDigit(UBYTE test_char);
ULONG str_hookRoutine(struct Hook *hook, struct SGWork *sgw, ULONG *msg);
void initHook(struct Hook *hook, ULONG (*ccode)());
VOID handleWindow(struct Vars *vars);

struct Library    *IntuitionBase;
struct Library    *UtilityBase;

#define SG_STRLEN     (44)
#define MYSTRGADWIDTH (200)
#define INIT_LATER     0

/* A border for the string gadget */
UWORD strBorderData[] = /* init elements 5 and 7 later (height adjust) */
    {
    0,0,  MYSTRGADWIDTH + 3,0,  MYSTRGADWIDTH + 3,INIT_LATER,
    0,INIT_LATER,    0,0,
    };
struct Border strBorder =
    {
    -2,-2, 1, 0,JAM1,5,strBorderData,NULL,
    };

/* We'll dynamically allocate/clear most structures, buffers */
struct Vars
    {
    struct Window       *sgg_Window;
    struct Gadget        sgg_Gadget;
    struct StringInfo    sgg_StrInfo;
    struct StringExtend  sgg_Extend;
    struct Hook          sgg_Hook;
    UBYTE                sgg_Buff[SG_STRLEN];
    UBYTE                sgg_WBuff[SG_STRLEN];
    UBYTE                sgg_UBuff[SG_STRLEN];
    };

/*   Main entry point.
**
** Open all required libraries, set-up the string gadget.
** Prepare the hook, open the sgg_Window and go...
*/
VOID main(int argc, char **argv)
{
struct Vars   *vars;
```

```
struct Screen   *screen;
struct DrawInfo *drawinfo;

if (IntuitionBase = OpenLibrary("intuition.library", 37L))
    {
    if (UtilityBase = OpenLibrary("utility.library", 37L))
        {
        /* get the correct pens for the screen. */
        if (screen = LockPubScreen(NULL))
            {
            if (drawinfo = GetScreenDrawInfo(screen))
                {
                vars = (struct Vars *)AllocMem(sizeof(struct Vars),MEMF_CLEAR);
                if (vars != NULL)
                    {
                    vars->sgg_Extend.Pens[0] = drawinfo->dri_Pens[FILLTEXTPEN];
                    vars->sgg_Extend.Pens[1] = drawinfo->dri_Pens[FILLPEN];
                    vars->sgg_Extend.ActivePens[0] = drawinfo->dri_Pens[FILLTEXTPEN];
                    vars->sgg_Extend.ActivePens[1] = drawinfo->dri_Pens[FILLPEN];
                    vars->sgg_Extend.EditHook = &(vars->sgg_Hook);
                    vars->sgg_Extend.WorkBuffer = vars->sgg_WBuff;

                    vars->sgg_StrInfo.Buffer = vars->sgg_Buff;
                    vars->sgg_StrInfo.UndoBuffer = vars->sgg_UBuff;
                    vars->sgg_StrInfo.MaxChars = SG_STRLEN;
                    vars->sgg_StrInfo.Extension = &(vars->sgg_Extend);

                    /* There should probably be a border around the string gadget.
                    ** As is, it is hard to locate when disabled.
                    */
                    vars->sgg_Gadget.LeftEdge = 20;
                    vars->sgg_Gadget.TopEdge = 30;
                    vars->sgg_Gadget.Width = MYSTRGADWIDTH;
                    vars->sgg_Gadget.Height = screen->RastPort.TxHeight;
                    vars->sgg_Gadget.Flags = GFLG_GADGHCOMP | GFLG_STRINGEXTEND;
                    vars->sgg_Gadget.Activation = GACT_RELVERIFY;
                    vars->sgg_Gadget.GadgetType = GTYP_STRGADGET;
                    vars->sgg_Gadget.SpecialInfo = &(vars->sgg_StrInfo);
                    vars->sgg_Gadget.GadgetRender = (APTR)&strBorder;
                    strBorderData[5] = strBorderData[7] =
                          screen->RastPort.TxHeight + 3;

                    initHook(&(vars->sgg_Hook), str_hookRoutine);

                    if (vars->sgg_Window = OpenWindowTags(NULL,
                            WA_PubScreen,       screen,
                            WA_Left,      21,   WA_Top,       20,
                            WA_Width,    500,   WA_Height,   150,
                            WA_MinWidth,  50,   WA_MaxWidth,  ~0,
                            WA_MinHeight, 30,   WA_MaxHeight, ~0,
                            WA_SimpleRefresh, TRUE,
                            WA_NoCareRefresh, TRUE,
                            WA_RMBTrap,       TRUE,
                            WA_IDCMP,         IDCMP_GADGETUP | IDCMP_CLOSEWINDOW,
                            WA_Flags,         WFLG_CLOSEGADGET | WFLG_NOCAREREFRESH |
                                              WFLG_DRAGBAR | WFLG_DEPTHGADGET |
                                              WFLG_SIMPLE_REFRESH,
                            WA_Title,         "String Hook Accepts HEX Digits Only",
                            WA_Gadgets,       &(vars->sgg_Gadget),
                            TAG_DONE))
                        {
                        handleWindow(vars);

                        CloseWindow(vars->sgg_Window);
                        }
                    FreeMem(vars,sizeof(struct Vars));
                    }
                FreeScreenDrawInfo(screen, drawinfo);
                }
            UnlockPubScreen(NULL, screen);
            }
        CloseLibrary(UtilityBase);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

```c
/*
** This is an example string editing hook, which shows the basics of
** creating a string editing function.  This hook restricts entry to
** hexadecimal digits (0-9, A-F, a-f) and converts them to upper case.
** To demonstrate processing of mouse-clicks, this hook also detects
** clicking on a character, and converts it to a zero.
**
** NOTE: String editing hooks are called on Intuition's task context,
** so the hook may not use DOS and may not cause Wait() to be called.
*/

ULONG str_hookRoutine(struct Hook *hook, struct SGWork *sgw, ULONG *msg)
{
UBYTE *work_ptr;
ULONG return_code;

/* Hook must return non-zero if command is supported.
** This will be changed to zero if the command is unsupported.
*/
return_code = ~0L;

if (*msg == SGH_KEY)
    {
    /* key hit -- could be any key (Shift, repeat, character, etc.) */

    /* allow only upper case characters to be entered.
    ** act only on modes that add or update characters in the buffer.
    */
    if ((sgw->EditOp == EO_REPLACECHAR) ||
        (sgw->EditOp == EO_INSERTCHAR))
        {
        /* Code contains the ASCII representation of the character
        ** entered, if it maps to a single byte.  We could also look
        ** into the work buffer to find the new character.
        **
        **     sgw->Code == sgw->WorkBuffer[sgw->BufferPos - 1]
        **
        ** If the character is not a legal hex digit, don't use
        ** the work buffer and beep the screen.
        */
        if (!IsHexDigit(sgw->Code))
            {
            sgw->Actions |= SGA_BEEP;
            sgw->Actions &= ~SGA_USE;
            }
        else
            {
            /* And make it upper-case, for nicety */
            sgw->WorkBuffer[sgw->BufferPos - 1] = ToUpper(sgw->Code);
            }
        }
    }
else if (*msg == SGH_CLICK)
    {
    /* mouse click
    ** zero the digit clicked on
    */
    if (sgw->BufferPos < sgw->NumChars)
        {
        work_ptr = sgw->WorkBuffer + sgw->BufferPos;
        *work_ptr = '0';
        }
    }
else
    {
    /* UNKNOWN COMMAND
    ** hook should return zero if the command is not supported.
    */
    return_code = 0;
    }

return(return_code);
}
```

```
/*
** This is a function which converts register-parameter
** hook calling convention into standard C conventions.
** It only works with SAS C 5.0+
**
** Without the fancy __asm stuff, you'd probably need to
** write this in assembler.
**
** You could conceivably declare all your C hook functions
** this way, and eliminate the middleman (you'd initialize
** the h_Entry field to your C function's address, and not
** bother with the h_SubEntry field).
**
** This is nice and easy, though, and since we're using the
** small data model, using a single interface routine like this
** (which does the necessary __saveds), it might
** actually turn out to be smaller to use a single entry point
** like this rather than declaring each of many hooks __saveds.
*/
ULONG __saveds __asm hookEntry(register __a0 struct Hook *hookptr,
    register __a2 void  *object,
    register __a1 void  *message)
{
return((*hookptr->h_SubEntry)(hookptr, object, message));
}


/*
** Initialize the hook to use the hookEntry() routine above.
*/
void initHook(struct Hook *hook, ULONG (*ccode)())
{
hook->h_Entry    = hookEntry;
hook->h_SubEntry = ccode;
hook->h_Data     = 0;   /* this program does not use this */
}

/*
** Process messages received by the sgg_Window.  Quit when the close gadget
** is selected.
*/
VOID handleWindow(struct Vars *vars)
{
struct IntuiMessage *msg;
ULONG  class;
USHORT code;

for (;;)
    {
    Wait(1L << vars->sgg_Window->UserPort->mp_SigBit);
    while (msg =
            (struct IntuiMessage *)GetMsg(vars->sgg_Window->UserPort))
        {
        /* Stash message contents and reply, important when message
        ** triggers some lengthy processing
        */
        class = msg->Class;
        code  = msg->Code;
        ReplyMsg((struct Message *)msg);

        switch (class)
            {
            case IDCMP_GADGETUP:
                /* if a code is set in the hook after an SGH_KEY
                ** command, where SGA_END is set on return from
                ** the hook, the code will be returned in the Code
                ** field of the IDCMP_GADGETUP message.
                */
                break;
            case IDCMP_CLOSEWINDOW:
                return;
                break;
            }
        }
    }
}
```

```
/*
** IsHexDigit()
**
** Return TRUE if the character is a hex digit (0-9, A-F, a-f)
*/
BOOL IsHexDigit(UBYTE test_char)
{
test_char = ToUpper(test_char);
if (((test_char >= '0') && (test_char <= '9')) ||
    ((test_char >= 'A') && (test_char <= 'F')))
    return(TRUE);
else
    return(FALSE);
}
```

CUSTOM GADGETS

Intuition also supports custom gadgets, where the application can supply to Intuition its own code to manage gadgets. This allows the creation of gadgets with behavior quite different from standard boolean, proportional, or string gadgets. For example, it would be possible to create a dial gadget, where the user could rotate the knob of a gadget. The code for a custom gadget needs to respond to various commands and requests from Intuition, such as "is this pixel in your hit-area?", "please go active" and "please go inactive".

Intuition has an object-oriented creation and delegation method called BOOPSI, that allows custom gadgets to be easily created, deleted, specialized from existing classes of custom gadget, and so on. See the Intuition chapter "BOOPSI" for details.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of Intuition gadgets. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 5-3: Functions for Intuition Gadgets

Function	Description
AddGadget()	Add a gadget to an open window or requester.
AddGList()	Add some gadgets to an open window or requester.
RemoveGadget()	Remove a gadget from an open window or requester.
RemoveGList()	Remove some gadgets from an open window or requester.
RefreshGadgets()	Refresh all gadgets for the window or requester.
RefreshGList()	Refresh some gadgets from the window or requester.
ModifyProp()	Change the values of an open proportional gadget.
NewModifyProp()	Optimized version of ModifyProp().
OnGadget()	Enable an open gadget.
OffGadget()	Disable an open gadget.
ActivateGadget()	Activate an open string gadget.
SetEditHook()	Change the global edit hook for string gadgets.

Chapter 6
INTUITION MENUS

Menus are command and option lists associated with an application window that the user can bring into view at any time. These lists provide the user with a simple way to access features of the application without having to remember or enter complex character-based command strings.

The Intuition menu system handles all of the menu display without intervention from the application. The program simply submits an initialized list of data structures to Intuition and waits for menu events.

This chapter shows how to set up menus that allow the user to choose from your program's commands and options.

About Menus

Intuition's menu system provides applications with a convenient way to group together and display the commands and options available to the user. In most cases menus consist of a fixed list of text choices however this is not a requirement. Items in the menu list may be either graphic images or text, and the two types can be freely used together. The number of items in a menu can be changed if necessary.

TYPES OF MENU CHOICES

Menu choices represent either actions or attributes. Actions are analogous to verbs. An action is executed and then forgotten. Actions include such things as saving and printing files, calculating values and displaying information on the program.

Attributes are analogous to adjectives. An attribute stays in effect until canceled. Attributes include such things as pen type, color, draw mode and numeric format.

For instance, in a word processor, menus could be used to control the following types of features:

- File loading and saving (action).
- Editing functions (action).
- Formatting preferences (attributes).
- Printing functions (action).
- Current font and style (attributes).

Menus can be set up such that some attribute items are mutually exclusive (selecting an attribute cancels the effects of one or more other attributes). For example, a drawing or graphics package may only allow one color to be active at a time--selecting a color cancels the previous active color.

The program can also allow a number of attributes to be in effect at the same time. A common example of this appears in most word processing programs, where the text style may be bold, italic or underlined. Selecting bold does not rule out italic or underlined, in fact, all three may be active at the same time.

THE MENU SYSTEM

To activate the menu system, the user presses the menu button (the right mouse button). This displays the menu bar in the screen's title area. The menu bar displays a list of topics (called menus) that have menu items associated with them (see figure). The menu bar and menu items only remain visible while the menu button is held down.

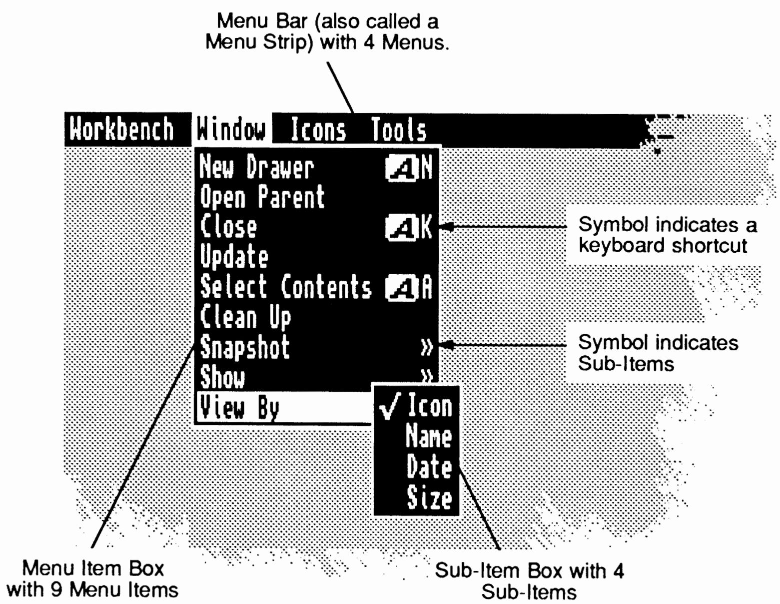


Figure 6-1: Screen with Menu Bar Displayed

When the mouse pointer is moved onto one of the menus in the menu bar, a list of menu items appears below the menu. The user can move the pointer within the list of menu items while holding down the menu button. A menu item will highlight when the pointer is over it and, if the item has a sub-item list, that list will be displayed.

The specific menu that is displayed belongs to the active window. Changing the active window will change the menu bar and the choices available to the user.

Unlike some other systems, the Amiga has no "standard menu" that appears in every menu bar. In fact, a window need not have any menus at all, thus holding down the mouse menu button does not guarantee the appearance of a menu bar. Although there is no "standard menu", Commodore does have a well-defined set of standards for menu design. These standards are covered in *The Amiga User Interface Style Guide* (also from Addison-Wesley).

Selecting Menu Items

To select a single menu item, the user releases the menu button when the pointer is over the desired item. Intuition can notify your program whenever the user makes a menu selection by sending an IDCMP message to your window's **UserPort**. Your application is then responsible for carrying out the action associated with the menu item selected. Action items lead to actions taken by the program while attribute items set values in the program for later reference.

Menu selection is restricted to the most subordinate item. Top level menus are never selected. A menu item can be selected as long as it has no sub-items, and a sub-item may always be selected. (Of course, disabled menu items and sub-items cannot be selected.) Intuition menus allow the user to select multiple items by:

- Pressing and releasing the select button (left mouse button) without releasing the menu button. This selects the item and keeps the menus active so that other items may be selected.

- Holding down both mouse buttons and sliding the pointer over several items. This is called drag selecting. All items highlighted while dragging are selected.

Drag selection, single selection with the select button and releasing the mouse button over an item can all be combined in a single operation. Any technique used to select a menu item is also available to select a menu sub-item.

Menu Item Imagery

Menu items can be graphic images or text. There is no conceptual difference between menus that display text and menus that display images, in fact, the two techniques may be used together. The examples in this chapter use text based menus to avoid the extra code required to define images.

When the user positions the pointer over an item, the item can be highlighted through a variety of techniques. These techniques include a highlighted box around the selected item, complementing the entire item and replacing the item with an alternate image or alternate text.

Attribute items can have an image rendered next to them, usually a checkmark, to indicate whether they are in effect or not. The checkmark is positioned to the left of the item. If the checkmark is present, the attribute is on. If not, the attribute is off.

On the right side of menu items, command key alternatives may be displayed. Command key alternatives allow the user to make menu selections with the keyboard instead of the mouse. This is done by holding down the right Amiga key and then pressing the single character command key alternative listed next to the menu item. Command key alternatives appear as a reverse video, fancy "A", followed by the single character command key.

Menu items or whole menus may be enabled or disabled. Disabling an item prevents the user from selecting it. Disabled items are ghosted (overwritten with a pattern of dots making the image less distinct) in order to distinguish them from enabled items. Menu help, a new feature of Release 2, allows the application to be notified when the user presses the help key at the same time the menu system is activated. This allows applications to provide a help feature for every item in its menus. Menu help may be requested on any level of a menu.

MENU LIMITATIONS

Menus are not layered so they lock the screen while they are displayed. While the screen is locked, applications cannot render graphics into that screen--any rendering will be suspended until the menus are no longer displayed.

Menus can only display a limited number of choices. Each window may have up to 31 menus, each menu may have up to 63 items, and each item may have up to 31 sub-items.

Menus always appear at the top of the screen and cannot be repositioned or sized by the user. Moving the pointer to the menu bar may be inconvenient or time consuming for the user. (This is why it is generally a good idea to provide keyboard alternatives for menu items.) If some application has a function that the user will be performing repeatedly, it may be better to use a series of gadgets in the window (or a separate window) rather than a menu item.

Alternatives to Menus

You may want to use a requester or a window as an alternative to menus. A requester can function as a "super menu" using gadgets to provide the commands and options of a menu but with fewer restrictions on their placement, size and layout. See the chapter entitled "Intuition Requesters and Alerts," for more information.

A window, also, could be substituted for a menu where an application has special requirements. Unlike menus, windows allow layered operations so that commands and options can be presented without forcing all other window output in the active screen to halt.

Windows may be sized, positioned and depth arranged. This positioning flexibility allows the user to make other parts of the screen and other windows visible while they are entering data or selecting operations. The ability to access or view other data may be important in the user's choice of actions or attributes. See the "Intuition Windows" chapter for more details.

Setting Up Menus

The application does not have to worry about handling the menu display. The menus are simply submitted to Intuition and the application waits for Intuition to send messages about the selection of menu items. These messages, along with the data in the menu structures, give the application all the information required for the processing of the user actions.

Menus can be set up with the GadTools library on systems running Release 2 or a later version of the OS. Since GadTools makes menu set up easier and handles much of the detail work of menu processing (including adjusting to the current font selection), it should be used whenever possible.

Under 1.3 (V34) and older versions of the OS, GadTools is not available. To set up menus that work with these older systems, you use the **Menu** and **MenuItem** structures. In general, for each menu in the menu bar, you declare one instance of the **Menu** structure. For each item or sub-item within a menu, you declare one instance of the **MenuItem** structure. Text-based menus like the kind used in this chapter require an additional **IntuiText** structure for each menu, menu item and sub-item. All these structures are defined in *<intuition/intuition.h>*.

The data structures used for menus are linked together to form a list known as a *menu strip*. For all the details of how the structures are linked and for listings of **Menu** and **MenuItem**, see the ''Menu Structures'' section later in this chapter.

SUBMITTING AND REMOVING MENU STRIPS

Once the application has set up the proper menu structures, linked them into a list and attached the list to a window, the menu system completely handles the menu display. The menu strip is submitted to Intuition and attached to the window by calling the function **SetMenuStrip()**.

```
BOOL SetMenuStrip( struct Window *window, struct Menu *menu );
```

SetMenuStrip() always returns TRUE. This function can also be used to attach a single menu strip to multiple windows by calling **SetMenuStrip()** for each window (see below).

Any menu strip attached to a window must be removed before the window is closed. To remove the menu strip, call **ClearMenuStrip()**.

```
void ClearMenuStrip( struct Window *window );
```

The menu example below demonstrates how to use these functions with a simple menu strip.

SIMPLE MENU EXAMPLE

Menu concepts are explained in great detail later in this chapter; for now though it may be helpful to look at an example. Here is a very simple example of how to use the Intuition menu system. The example shows how to set up a menu strip consisting of a single menu with five menu items. The third menu item in the menu has two sub-items.

The example works with all versions of the Amiga OS however it assumes that the Workbench screen is set up with the the Topaz 8 ROM font. If the font is different, the example will exit immediately since the layout of the menus depends on having a monospaced font with 8 x 8 pixel characters.

```
;/* simplemenu.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 simplemenu.c
Blink FROM LIB:c.o,simplemenu.o TO simplemenu LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** simplemenu.c: how to use the menu system with a window under all OS versions.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/text.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/*  These values are based on the ROM font Topaz8. Adjust these  */
/*  values to correctly handle the screen's current font.        */
#define MENWIDTH  (56+8)  /* Longest menu item name * font width */
                          /* + 8 pixels for trim                 */
#define MENHEIGHT (10)    /* Font height + 2 pixels              */

struct Library *GfxBase;
struct Library *IntuitionBase;

/* To keep this example simple, we'll hard-code the font used for menu */
/* items.  Algorithmic layout can be used to handle arbitrary fonts.   */
/* Under Release 2, GadTools provides font-sensitive menu layout.      */
/* Note that we still must handle fonts for the menu headers.          */
struct TextAttr Topaz80 =
{
    "topaz.font", 8, 0, 0
};

struct IntuiText menuIText[] =
{
    { 0, 1, JAM2, 0, 1, &Topaz80, "Open...",  NULL },
    { 0, 1, JAM2, 0, 1, &Topaz80, "Save",     NULL },
    { 0, 1, JAM2, 0, 1, &Topaz80, "Print \273",  NULL },
    { 0, 1, JAM2, 0, 1, &Topaz80, "Draft",    NULL },
    { 0, 1, JAM2, 0, 1, &Topaz80, "NLQ",      NULL },
    { 0, 1, JAM2, 0, 1, &Topaz80, "Quit",     NULL }
};

struct MenuItem submenu1[] =
{
    { /* Draft  */
    &submenu1[1], MENWIDTH-2,  -2 ,          MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[3], NULL, NULL, NULL, NULL
    },
    { /* NLQ    */
    NULL,         MENWIDTH-2, MENHEIGHT-2, MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[4], NULL, NULL, NULL, NULL
    }
};

struct MenuItem menu1[] =
{
    { /* Open... */
    &menu1[1], 0, 0,           MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[0], NULL, NULL, NULL, NULL
    },
```

```
    { /* Save    */
    &menu1[2], 0,  MENHEIGHT ,  MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[1], NULL, NULL, NULL, NULL
    },
    { /* Print   */
    &menu1[3], 0, 2*MENHEIGHT , MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[2], NULL, NULL, &submenu1[0] , NULL
    },
    { /* Quit    */
    NULL, 0, 3*MENHEIGHT , MENWIDTH, MENHEIGHT,
    ITEMTEXT | MENUTOGGLE | ITEMENABLED | HIGHCOMP,
    0, (APTR)&menuIText[5], NULL, NULL, NULL, NULL
    },
};

/* We only use a single menu, but the code is generalizable to */
/* more than one menu.                                         */
#define NUM_MENUS 1

STRPTR menutitle[NUM_MENUS] =  {   "Project"   };

struct Menu menustrip[NUM_MENUS] =
{
    {
    NULL,                    /* Next Menu          */
    0, 0,                    /* LeftEdge, TopEdge, */
    0, MENHEIGHT,            /* Width, Height,     */
    MENUENABLED,             /* Flags              */
    NULL,                    /* Title              */
    &menu1[0]                /* First item         */
    }
};

struct NewWindow mynewWindow =
{
40,40, 300,100, 0,1, IDCMP_CLOSEWINDOW | IDCMP_MENUPICK,
WFLG_DRAGBAR | WFLG_ACTIVATE | WFLG_CLOSEGADGET, NULL,NULL,
"Menu Test Window", NULL,NULL,0,0,0,0,WBENCHSCREEN
};

/* our function prototypes */
VOID handleWindow(struct Window *win, struct Menu *menuStrip);

/*      Main routine.         */
/*                            */
VOID main(int argc, char **argv)
{
struct Window *win=NULL;
UWORD left, m;

/* Open the Graphics Library */
GfxBase = OpenLibrary("graphics.library",33);
if (GfxBase)
    {
    /* Open the Intuition Library */
    IntuitionBase = OpenLibrary("intuition.library", 33);
    if (IntuitionBase)
        {
        if ( win = OpenWindow(&mynewWindow) )
            {
            left = 2;
            for (m = 0; m < NUM_MENUS; m++)
                {
                menustrip[m].LeftEdge = left;
                menustrip[m].MenuName = menutitle[m];
                menustrip[m].Width = TextLength(&win->WScreen->RastPort,
                    menutitle[m], strlen(menutitle[m])) + 8;
                left += menustrip[m].Width;
                }
            if (SetMenuStrip(win, menustrip))
                {
                handleWindow(win, menustrip);
                ClearMenuStrip(win);
                }
```

```
            CloseWindow(win);
            }
        CloseLibrary(IntuitionBase);
        }
    CloseLibrary(GfxBase);
    }
}

/*
**   Wait for the user to select the close gadget.
*/
VOID handleWindow(struct Window *win, struct Menu *menuStrip)
{
struct IntuiMessage *msg;
SHORT done;
ULONG class;
UWORD menuNumber;
UWORD menuNum;
UWORD itemNum;
UWORD subNum;
struct MenuItem *item;

done = FALSE;
while (FALSE == done)
    {
    /* we only have one signal bit, so we do not have to check which
    ** bit broke the Wait().
    */
    Wait(1L << win->UserPort->mp_SigBit);

    while ( (FALSE == done) &&
            (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        class = msg->Class;
        if(class == IDCMP_MENUPICK)   menuNumber = msg->Code;

        switch (class)
            {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            case IDCMP_MENUPICK:
                while ((menuNumber != MENUNULL) && (!done))
                    {
                    item = ItemAddress(menuStrip, menuNumber);

                    /* process this item
                    ** if there were no sub-items attached to that item,
                    ** SubNumber will equal NOSUB.
                    */
                    menuNum = MENUNUM(menuNumber);
                    itemNum = ITEMNUM(menuNumber);
                    subNum  = SUBNUM(menuNumber);

                    /* Note that we are printing all values, even things
                    ** like NOMENU, NOITEM and NOSUB.  An application should
                    ** check for these cases.
                    */
                    printf("IDCMP_MENUPICK: menu %d, item %d, sub %d\n",
                        menuNum, itemNum, subNum);

                    /* This one is the quit menu selection...
                    ** stop if we get it, and don't process any more.
                    */
                    if ((menuNum == 0) && (itemNum == 4))
                        done = TRUE;

                    menuNumber = item->NextSelect;
                    }
                break;
            }
        ReplyMsg((struct Message *)msg);
        }
    }
}
```

DISABLING MENU OPERATIONS

If an application does not use menus at all, it may set the WFLG_RMBTRAP flag, which allows the program to trap right mouse button events for its own use.

By setting the WFLG_RMBTRAP flag with the WA_Flags tag when the window is opened, the program indicates that it does not want any menu operations at all for the window. Whenever the user presses the right button while this window is active, the program will receive right button events as normal IDCMP_MOUSEBUTTONS events.

CHANGING MENU STRIPS

Direct changes to a menu strip attached to a window may be made only after the menu strip has been removed from the window. Use the **ClearMenuStrip()** function to remove the menu strip. It may be added back to the window after the changes are complete.

Major changes include such things as adding or removing menus, items and sub-items; changing text or image data; and changing the placement of the data. These changes require the system to completely re-layout the menus.

An additional function, **ResetMenuStrip()**, is available to let the application make small changes to the menus without the overhead of **SetMenuStrip()**. Only two things in the menu strip may be changed before a call to **ResetMenuStrip()**, they are: changing the CHECKED flag to turn checkmarks on or off, and changing the ITEMENABLED flag to enable/disable menus, items or sub-items.

```
BOOL ResetMenuStrip( struct Window *window, struct Menu *menu );
```

ResetMenuStrip() is called in place of **SetMenuStrip()**, and may only be called on menus that were previously initialized with a call to **SetMenuStrip()**. As with **SetMenuStrip()**, the menu strip must be removed from the window before calling **ResetMenuStrip()**. Note that the window used in the **ResetMenuStrip()** call does not have to be the same window to which the menu was previously attached. The window, however, must be on a screen of the same mode to prevent the need for recalculating the layout of the menu.

If the application wishes to attach a different menu strip to a window that already has an existing menu strip, the application must call **ClearMenuStrip()** before calling **SetMenuStrip()** with the new menu strip.

The flow of events for menu operations should be:

1. **OpenWindowTagList()**.
2. **SetMenuStrip()**.
3. Zero or more iterations of **ClearMenuStrip()** and **SetMenuStrip()/ResetMenuStrip()**.
4. **ClearMenuStrip()**.
5. **CloseWindow()**.

SHARING MENU STRIPS

A single menu strip may be attached to multiple windows in an application by calling **SetMenuStrip()** for each window. All of the windows must be on the same screen for this to work. Since menus are always associated with the active window on a given screen, and since only one window may be active on a screen at a time, only one window may display the shared menu strip at any given time.

When multiple windows share a single menu strip, they will all "see" the same state of the menus, that is, changes made to the menu strip from one window will still exist when a new window is activated. If the application wishes to share menu strips but to have a different flag and enabled status for each window, the program may watch IDCMP_ACTIVEWINDOW for the windows and modify the menu strip to match the active window's requirements at that point. In addition, the application must also set IDCMP_MENUVERIFY to insure that the user can't access the menus of a newly activated window before the application can process the IDCMP_ACTIVEWINDOW message.

ResetMenuStrip() may also be used to set the menus for the multiple windows as long as **SetMenuStrip()** is used first to attach the menu strip to any one window and no major changes are made to the menu strip before the calls to **ResetMenuStrip()** on subsequent windows.

MENU SELECTION MESSAGES

An input event is generated every time the user activates the menu system, either by pressing the mouse menu button or its keyboard equivalent (right Amiga Alt), or entering an appropriate command key sequence. The program receives a message of type IDCMP_MENUPICK detailing which menu items or sub-items were selected. Even if the user activates the menu system without selecting a menu item or sub-item, an event is generated.

Multi-Selection of Menu Items

Each activation of the menu system generates only a single event. The user may select none, one or many items using any of the selection techniques described above; still, only one event is sent.

The program finds out whether or not multiple items have been chosen by examining the field called **NextSelect** in the **MenuItem** structure. The selected items are chained together through this field. This list is only valid until the user next activates the menu system, as the items are chained together through the actual **MenuItem** structures used in the menu system. If the user reselects an item, the **NextSelect** field of that item will be overwritten.

In processing the menu events, the application should first take the appropriate action for the item selected by the user, then check the **NextSelect** field. If the number there is equal to the constant MENUNULL, there is no next selection. However, if it is not equal to MENUNULL, the user has selected another option after this one. The program should process the next item as well, by checking its **NextSelect** field, until it finds a **NextSelect** equal to MENUNULL.

The following code fragment shows the correct way to process a menu event:

```
struct IntuiMessage *msg;
struct Menu *menuStrip;
UWORD menuNumber;
struct MenuItem *item;

menuNumber = msg->Code;

while (menuNumber != MENUNULL)
    {
    item = ItemAddress(menuStrip, menuNumber);

    /* process this item */

    menuNumber = item->NextSelect;
    }
```

Intuition specifies which item or sub-item was selected in the IDCMP_MENUPICK event by using a shorthand code known as a *menu number*. Programs can locate the **MenuItem** structure that corresponds to a given menu number by using the **ItemAddress**() function. This function translates a menu number into a pointer to the **MenuItem** structure represented by the menu number.

```
struct MenuItem *ItemAddress( struct Menu *menuStrip, unsigned long menuNumber );
```

This allows the application to gain access to the **MenuItem** structure and to correctly process multi-select events. Again, when the user performs multiple selection, the program will receive only one message of class IDCMP_MENUPICK. For the program to behave correctly, it must pay attention to the **NextSelect** field of the **MenuItem**, which will lead to the other menu selections.

There may be some cases in an application's logical flow where the selection of a menu item voids any further menu processing. For instance, after processing a "quit" menu selection, the application will, in general, ignore all further menu selections.

MENU NUMBERS

The menu numbers Intuition provides in the IDCMP_MENUPICK messages, describe the ordinal position of the menu in the linked list of menus, the ordinal position of the menu item beneath that menu, and (if applicable) the ordinal position of the sub-item beneath that menu item. Ordinal means the successive number of the linked items, in this case, starting from 0.

To determine which menus and menu items (sub-items are special cases of menu items) were selected, use the following macros:

Table 6-1: Macros Used with Intuition Menus

MENUNUM(num)	Extracts the ordinal menu number from **num**.
ITEMNUM(num)	Extracts the ordinal item number from **num**.
SUBNUM(num)	Extracts the ordinal sub-item number from **num**.

MENUNULL is the constant describing "no menu selection made." Likewise, NOMENU, NOITEM, and NOSUB describe the conditions "no menu chosen," "no item chosen" and "no sub-item chosen."

For example:

```
if (menuNumber == MENUNULL)
    /* no menu selection was made */ ;
else
    {
    /* if there were no sub-items attached to that item,
    ** SubNumber will equal NOSUB.
    */
    menuNum = MENUNUM(menuNumber);
    itemNum = ITEMNUM(menuNumber);
    subNum  = SUBNUM(menuNumber);
    }
```

The menu number received by the program is always set to either MENUNULL or a valid menu selection. If the menu number represents a valid selection, it will always have at least a menu number and a menu item number. Users can never select the menu text itself, but they always select at least an item within a menu. Therefore, the program always gets at least the menu selected and the menu item selected. If the menu item selected has a sub-item, a sub-item number will also be received.

Just as it is not possible to select an entry in the menu bar, it is not possible to select a menu item that has attached sub-items. The user must select one of the options in the sub-item list before the program hears about the action as a valid menu selection.

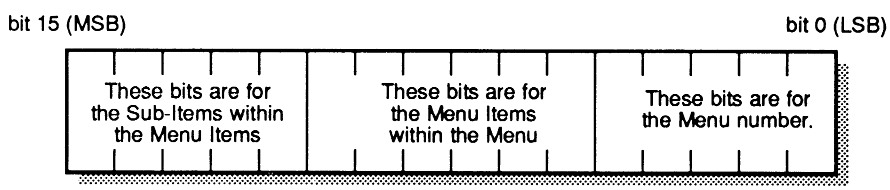
> *Help Is Available.* The restrictions on what can be selected do not apply to IDCMP_MENUHELP messages. Using menu help, a user can select any component of a menu, including the menu header itself.

How Menu Numbers Really Work

The following is a description of how menu numbers really work. It should clarify why there are some restrictions on the number of menu components Intuition allows. Programs should not rely on the information given here to access menu number information directly though. Always use the Intuition menu macros to handle menu numbers.

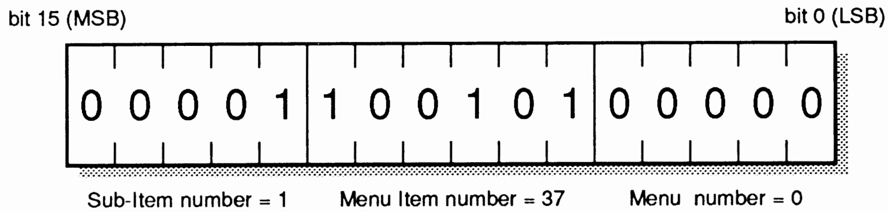
For convenience you should use the menus supplied. For example, to extract the item number from the menu number, call the macro **ITEMNUM**(menu_number); to construct a menu number, call the macro **FULLMENUNUM**(menu, item, sub). See the section at the end of this chapter for a more complete description of the menu number macros.

Menu numbers are 16-bit numbers with 5 bits used for the menu number, 6 bits used for the menu item number, and 5 bits used for the sub-item number. The three numbers only have meaning when used together to determine the position of the item or sub-item selected.



The value "all bits on" means that no selection of this particular component was made. MENUNULL actually equals "no selection of any of the components was made" so MENUNULL always equals "all bits of all components on."

For example, suppose that the program gets back the menu number (in hexadecimal) 0x0CA0. In binary that equals:



Again, the application should not examine these numbers directly. Use the macros described above to ensure proper menu handling.

HELP KEY PROCESSING IN MENUS

If the window is opened with the WA_MenuHelp tag, then user selection of the help key while menus are displayed will be detected. This tag is only available under V37 and later.

When the user presses the Help key while using the menu system, the menu selection is terminated and an IDCMP_MENUHELP event is sent. The IDCMP_MENUHELP event is sent in place of the IDCMP_MENUPICK event, not in addition to it. IDCMP_MENUHELP never come as multi-select items, and the event terminates the menu processing session.

The routine that handles the IDCMP_MENUHELP events must be very careful--it can receive menu information that is impossible under IDCMP_MENUPICK. IDCMP_MENUHELP events may be sent for any menu, item or sub-item in the menu strip, regardless of its state or position in the list. The program may receive events for items that are disabled or ghosted. IDCMP_MENUHELP events may send the menu header number alone, or the menu and item numbers, or all three components regardless of the items linked to the selected menu or item. This is done because it is reasonable for a user to request help in a disabled item or a menu header. If the user requests menu help on a disabled menu item or sub-item, try to explain to the user why that item is disabled and what steps are necessary to enable it. For instance, pressing help while a menu header is highlighted will trigger an IDCMP_MENUHELP event with a code that has a valid number for the menu, then NOITEM and NOSUB (IDCMP_MENUPICK would receive MENUNULL in this case.)

The application should not take the action indicated by the IDCMP_MENUHELP event, it should provide the user with a description of the use or status of that menu. The application should never step through the **NextSelect** chain when it receives a IDCMP_MENUHELP event.

MENU LAYOUT

The Amiga allows great flexibility in the specification of fonts for the display. Default fonts are chosen by the user to suit their particular requirements and display resolution. The application should, where possible, use one of the preferred fonts.

If the application did not open its own screen and completely specify the font for that screen, it must perform dynamic menu layout. This is because the **Menu** structure does not specify font. The menu header always uses the screen font and the program should update the size and position of these items at runtime to reflect the font.

The font for menu items may be specified in the **MenuItem** structure, allowing the programmer to hard code values for the font, size and position of these items. This is not recommended. A specific font, while ideal on one system, may be less than ideal on another display type. Use the preferred font wherever possible.

If the application does its own menu layout, it must use great care to handle the font in the menu strip and the font in each item or sub-item. The code should also keep items from running off the edges of the screen.

See the description of **ItemFill** in the section ''MenuItem Structure'' below for information on the positioning of multiple **IntuiText** or **Image** structures within the menu item.

Applications should use the GadTools library menu layout routines whenever possible, rather than performing their own layout. See the chapter on the ''GadTools Library'' for more details.

ABOUT MENU ITEM BOXES

The item box is the rectangle containing the menu items or sub-items.

The size and location of the item or sub-item boxes is not directly described by the application. Instead, the size is indirectly described by the placement of items and sub-items. When presented with a menu strip, Intuition first calculates the minimum size box required to hold the items. It then adjusts the box to ensure the menu display conforms to certain design philosophy constraints for items and sub-items.

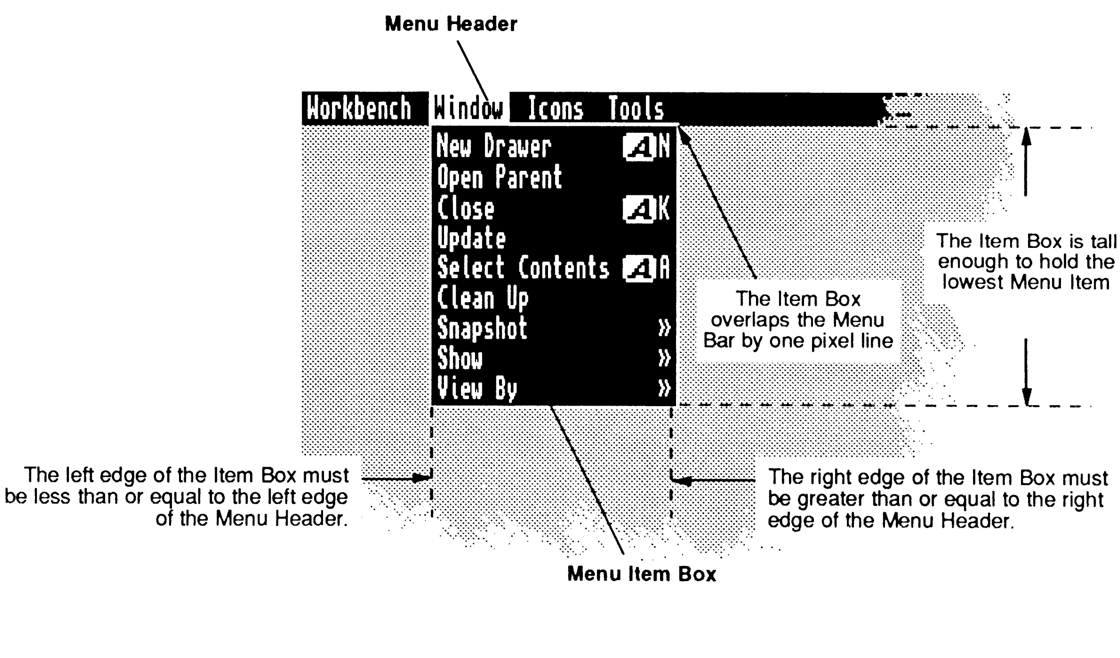


Figure 6-2: Example Item Box

The item box must start no further right than the leftmost position of the menu header's select box. It must end no further left than the rightmost position of the menu header's select box. The top edge of each item box must overlap the screen's title bar by one line. Each sub-item box must overlap its item's select box somewhere.

> *Always Overlap.* If your application is designed to work on systems prior to V37, do not leave space between sub-items in a sub-item list. This may cause flickering as the pointer moves off one item into the gap between the items. Even a single line between the items may cause flickering. This flickering is eliminated starting with V37.

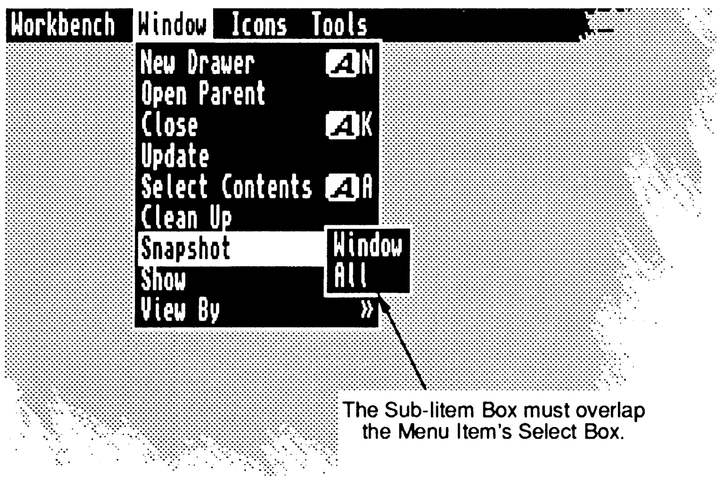


Figure 6-3: Example Sub-item Box

ATTRIBUTE ITEMS AND THE CHECKMARK

Attribute items are items that have two states: selected and unselected. In the menu system, these items are often represented as items with checkmarks. If the checkmark is visible, then the item (or attribute) is selected. Otherwise, the attribute is not selected.

Checkmarked items (attributes) may be toggle selected or mutually exclusive. Selecting a toggle selected item toggles its state--if it was selected, it will become unselected; and if it was unselected, it will become selected. Selecting a mutually exclusive item puts it in the selected state, while possibly clearing one or more other items, where it remains until it is cleared by the selection of some other item.

A menu item is specified as a checkmark item by setting the CHECKIT flag in the **Flags** variable of the item's **MenuItem** structure.

The program can initialize the state of the checkmark (checked or not) by presetting the item's CHECKED flag. If this flag is set when the menu strip is submitted to Intuition, then the item is considered selected and the checkmark will be drawn.

The program can use the default Intuition checkmark or provide a custom checkmark for the menus. To use a custom checkmark, the application must provide a pointer to the image with the WA_Checkmark tag when the window is opened. See the chapter "Intuition Windows" for details about supplying a custom checkmark.

The application must provide sufficient blank space at the left edge of the select box for the checkmark imagery. Constants are provided to standardize the space reserved in the menu for the checkmark. LOWCHECKWIDTH gives the amount of space required for checkmarks on low resolution screens and CHECKWIDTH gives space for all other screens.

These constants specify the space required by the default checkmarks (with a bit of space for aesthetic purposes). If the image would normally be placed such that the **LeftEdge** of the image without the checkmark is 5, the image should start at 5 + CHECKWIDTH if CHECKIT is set. Also, the select box must be made CHECKWIDTH wider than it would be without the checkmark. It is generally accepted on the Amiga that only checkmarked items should be indented by the size of the checkmark, other items are left justified within their column.

TOGGLE SELECTION

Some of the checkmarked menu items may be of the toggle select type. Each time the user accesses such an item, it changes state, selected or unselected. To make an attribute item toggle select, set both the CHECKIT and the MENUTOGGLE flags for that menu item. Of course, the CHECKED flag may be preset to the desired initial state.

MUTUAL EXCLUSION

Mutual exclusion allows the selection of an item to cause other items to become unselected.

For example, for a list of menu items describing the available sizes for a font, the selection of any size could unselect all other sizes. Use the **MutualExclude** variable in the **MenuItem** structure to specify other menu items to be excluded when the user selects an item. Exclusion also depends upon the CHECKED and CHECKIT flags of the **MenuItem**, as explained below.

- If CHECKED is not set, then this item is available to be selected. If the user selects this item, the CHECKED flag is set and the checkmark will be drawn to the left of the item.
- If the item selected has bits set in the **MutualExclude** field, the CHECKED flag is examined in the excluded items. If any item is currently CHECKED, its checkmark is erased, and its CHECKED flag is cleared.
- Mutual exclusion pertains only to items that have the CHECKIT flag set. Attempting to exclude items that do not have the CHECKIT flag set has no effect.

Keep track of deselected items. It is up to the program to track internally which excluded items have been deselected. See the section ''Enabling and Disabling Menus and Menu Items'' below for more information.

In the **MutualExclude** field, bit 0 refers to the first item in the item list, bit 1 to the second, bit 2 to the third, and so on.

In the text style example described above, selecting plain excludes any other style. The **MutualExclude** fields of the four items would look like this:

Plain	0xFFFE
Bold	0x0001
Italic	0x0001
Underline	0x0001

''Plain'' is the first item on the list. It excludes all items except the first one. All of the other items exclude only the first item, so that bold, underlined text may be selected, while bold, plain text may not.

MANAGING THE STATE OF CHECKMARKS

To correctly handle checkmarked menu items, from time to time the application will need to read the CHECKED bit of its CHECKIT **MenuItems**. It is not adequate to infer which items are checked by tracking what their state must have become. There are several reasons for this (although it's not important to understand the details; just the implication):

- Using multi-selection of menus, the user can toggle the state of a MENUTOGGLE item several times, yet the application will receive only a single IDCMP_MENUPICK event, and that item will only appear once one the **NextSelect** chain.

- When the user selects a mutually exclusive menu item, the IDCMP_MENUPICK event refers to that item, but Intuition doesn't notify your application of other items that may have been deselected through mutual exclusion.

- Prior to V36, unusually complex multi-selection operations could orphan menu selections. That is to say, some items that were selected may not even appear on the **NextSelect** chain. If such an item had a checkmark, the state of that checkmark could nevertheless have changed.

- For complex multi-selection operations, the **NextSelect** chain will not be in select-order (a side-effect of the fact that the same **MenuItem** cannot appear twice in the same **NextSelect** chain combined with the fix to the orphaning problems mentioned above). With certain mutual exclusion arrangements, it is impossible to predict the state of the checkmarks.

- If the user begins multi-selection in the menus and hits several checkmarked items, but then presses the help key, the application will receive an IDCMP_MENUHELP message. No IDCMP_MENUPICK message will have been sent. Thus, some checkmark changes could have gone unnoticed by the application.

It is legal to examine the CHECKED state of a **MenuItem** while that **MenuItem** is still attached to a window. It is unnecessary to first call **ClearMenuStrip()**.

COMMAND KEY SEQUENCES

A command key sequence is an event generated when the user holds down one of the Amiga keys (the ones with the fancy A) and presses one of the normal alphanumeric keys at the same time. There are two different command or Amiga keys, commonly known as the left Amiga key and the right Amiga key.

Menu command key sequences are combinations of the right Amiga key with any alphanumeric character, and may be used by any program. These sequences must be accessed through the menu system. Command key sequences using the left Amiga key cannot be associated with menu items.

Menu command key sequences, like the menus themselves, are only available for a window while that window is active. Each window may control these keys by setting keyboard shortcuts in the menu item structures which make up the window's menu strip.

If the user presses a command key sequence that is associated with one of the menu items, Intuition will send the program an event that is identical to the event generated by selecting that menu item with the mouse. Many users would rather keep their hands on the keyboard than use the mouse to make a menu selection when accessing often repeated selections. Menu command key sequences allow the program to provide shortcuts to the user who prefers keyboard control.

A command key sequence is associated with a menu item by setting the COMMSEQ flag in the **Flags** variable of the **MenuItem** structure and by placing the ASCII character (upper or lower case) that is to be associated with the sequence into the **Command** variable of the **MenuItem** structure.

Command keys are not case sensitive and they do not repeat. Command keys are processed through the keymap so that they will continue to work even if the key value is remapped to another position. International key values are supported as long as they are accessible without using the Alt key (right-Amiga-Alt maps to the right mouse button on the mouse).

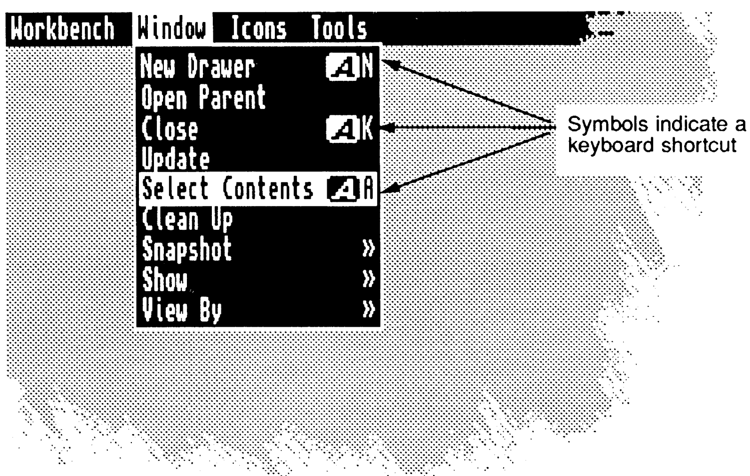


Figure 6-4: Menu Items with Command Key Shortcuts

When items have alternate key sequences, the menu boxes show a special Amiga key glyph rendered roughly one character span plus a few pixels from the right edge of the menu select box. The command key used with the Amiga key is displayed immediately to the right of the Amiga key image, at the rightmost edge of the menu select box (see figure).

Space must be provided at the right edge of the select box for the Amiga key imagery and for the actual command character. Leave COMMWIDTH pixels on high resolution screens, and LOWCOMMWIDTH pixels on low resolution screens. The character's width may be calculated with the graphics library **TextLength()** call. In general, each column of items should leave enough room for the widest command character plus the width of the Amiga key imagery.

ENABLING AND DISABLING MENUS AND MENU ITEMS

Disabling menu items makes them unavailable for selection by the user.

Disabled menus and menu items are displayed in a ghosted fashion; that is, their imagery is overlaid with a faint pattern of dots, making it less distinct.

Enabling or disabling a menu or menu item is always a safe procedure, whether or not the user is currently using the menus. Of course, by the time you have disabled the item, the user may have already selected it. Thus, the program may receive a IDCMP_MENUPICK message for that item, even though it considers the item disabled. The program should be prepared to handle this case and ignore items that it knows are already disabled. This implies that the program must track internally which items are enabled and which are disabled.

The **OffMenu()** and **OnMenu()** functions may be used to enable or disable items while a menu strip is attached to the window.

```
void OffMenu( struct Window *window, unsigned long menuNumber );
void OnMenu( struct Window *window, unsigned long menuNumber );
```

These routines check if the user is currently using the menus and whether the menus need to be redrawn to reflect the new states. If the menus are currently in use, these routines wait for the user to finish before proceeding.

If the item component referenced by **menuNumber** equals NOITEM, the entire menu will be disabled or enabled. If the item component equates to an actual component number, then that item will be disabled or enabled. Use the macros defined below for the construction of menu numbers from their component parts.

The program can enable or disable whole menus, just the menu items, or just single sub-items.

- To enable or disable a whole menu, set the item component of the menu number to NOITEM. This will enable or disable all items and any sub-items for that menu.
- To enable or disable a single item and all sub-items attached to that item, set the item component of the menu number to the item's ordinal number. If the item has a sub-item list, set the sub-item component of the menu number to NOSUB. If the item has no sub-item list, the sub-item component of the menu number is ignored.
- To enable or disable a single sub-item, set the item and sub-item components appropriately.

It is also legal to remove the menu strip from each window that it is attached to (with **ClearMenuStrip()**) change the ITEMENABLED or MENUENABLED flag of one or more **Menu** or **MenuItem** structures and add the menu back using **ResetMenuStrip()** (in V36 or higher) or **SetMenuStrip()** (in any version of the OS).

INTERCEPTING NORMAL MENU OPERATIONS

IDCMP_MENUVERIFY gives the program the opportunity to react before menu operations take place and, optionally, to cancel menu operations. Menus may be completely disabled by removing the menu strip with a call to **ClearMenuStrip()**.

A Warning on the MENUSTATE Flag

The MENUSTATE flag is set by Intuition in **Window.Flags** when the menus of that window are in use. Beware: in typical event driven programming, such a state variable is not on the same timetable as the application's input message handling, and should not be used to draw profound conclusions in any program. Use IDCMP_MENUVERIFY to synchronize with the menu handling.

Menu Verify

Menu verify is one of the Intuition verification capabilities that allow an application to ensure that it is prepared for some action taken by Intuition before that action takes place. Through menu verify, Intuition allows all windows in a screen to verify that they are prepared for menu operations before the operations begin. In general, use menu verify if the program is doing something special to the display of a custom screen, and needs to be sure the operation has completed before menus are rendered.

Any window can access the menu verify feature by setting the IDCMP_MENUVERIFY flag with the WA_IDCMP tag when opening the window. When menus are activated in a screen which contains at least one window with IDCMP_MENUVERFIY set, menu operations will not proceed until all windows with the menu verify flag set reply to the notification or until the last message times out. The specific menu verify protocol is described below.

In any case, it is vital that the application know when menu operations terminate, for only then does it have control of the screen again. For the active window, this is typically detected by watching for an IDCMP_MENUPICK message. If the program cancels the menu operations (MENUCANCEL), then it will instead receive an IDCMP_MOUSEBUTTONS message with code equal to MENUUP. Inactive windows will always receive IDCMP_MOUSEBUTTONS message with code equal to MENUUP.

The active window is given special menu verify treatment. It receives the menu verify message before any other window and has the option of canceling menu operations altogether. This could be used, for instance, to examine where the user has positioned the mouse when the right button was pressed. For example, the application may choose to allow normal menu operations to proceed only when the pointer is in the menu bar area. When the pointer is below the menu bar, then the application can choose to interpret the menu verify message as a right button event for some non-menu purpose.

The program can tell if it is the active window for the verify event by examining the **Code** field of the IDCMP_MENUVERIFY message. If the **Code** field is equal to MENUWAITING, this window is not active and Intuition is simply waiting for verification that menu operations may continue. However, if the **Code** field is equal to MENUHOT, this window is active and it determines if menu operations should proceed.

If it wishes menu operations to proceed, the active window should reply to the IDCMP_MENUVERIFY message without changing any values. To cancel the menu operation, change the code field of the message to MENUCANCEL before replying to the message.

When the active window cancels the menu operation it will be sent an IDCMP_MOUSEBUTTONS message with code equal to MENUUP. In general, the window will not then receive an IDCMP_MENUPICK event as it cancelled the operation. However, the system should be prepared to handle an IDCMP_MENUPICK message with code equal to MENUNULL as one may be sent if the user releases the mouse button before the window replies to the message.

The system takes no action on screen until the active window either replies to the menu verify event or the event times out. If the active window replies to the event in time and does not cancel the menu operation, Intuition will then move the screen title bar layer to the front, display the menu strip and notify all inactive menu verify windows of the operation. Layers will not be locked and the actual menus will not be swapped in until all these inactive windows reply to the message or time out. The inactive windows may not cancel the menu operation.

If the user releases the menu button before the active window replies to the menu verify message, the menu operation will be cancelled and the active window will be sent an IDCMP_MOUSEBUTTONS message with code equal to MENUUP. When the active window finally replies to the message, it will receive an IDCMP_MENUPICK message with code equal to MENUNULL.

If the event times out before the active window replies to the message, it will immediately be sent an IDCMP_MENUPICK message with code equal to MENUNULL. Then, when the user releases the menu button, the program will receive an IDCMP_MOUSEBUTTONS message with code equal to MENUUP.

If an inactive window receives an IDCMP_MENUVERIFY message, it will always receive an IDCMP_MOUSEBUTTONS message with code equal to MENUUP when the menu operations are completed.

> *About Double-Menu Requesters.* The processing described above becomes more complicated when double-menu requester processing is introduced. If an application chooses to use a double-menu requester in a window with IDCMP_MENUVERIFY set, it should be aware that odd message combinations are possible. For instance, it is possible to receive only an IDCMP_MENUVERIFY event with no following IDCMP_MOUSEBUTTONS event or IDCMP_MENUPICK event. Applications should avoid using double menu requesters if possible.

Shortcuts and IDCMP_MENUVERIFY

The idea behind IDCMP_MENUVERIFY is to synchronize the program with Intuition's menu handling sessions. The motive was to allow a program to arbitrate access to a custom screen's bitmap, so that Intuition would not render menus before the application was prepared for them.

Some programs use IDCMP_MENUVERIFY to permit them to intercept the right mouse button for their own purposes. Other programs use it to delay menu operations while they recover from unusual events such as illegible colors of the screen or double buffering and related **ViewPort** operations.

Menu shortcut keystrokes, for compatibility, also respect IDCMP_MENUVERIFY. They are always paired with an IDCMP_MENUPICK message so that the program knows the menu operation is over. This is true even if the menu event is cancelled.

IDCMP_MENUVERIFY and Deadlock

The program may call **ModifyIDCMP()** to turn IDCMP_MENUVERIFY and the other VERIFY IDCMP options off. It is important that this be done each and every time that the application is directly or indirectly waiting for Intuition, since Intuition may be waiting for the application, but not watching the window message port for IDCMP_MENUVERIFY events. The program cannot wait for a gadget or mouse event without checking also for any IDCMP_MENUVERIFY event messages that may require program response.

The most common problem area is System Requesters (**AutoRequest()** and **EasyRequest()**). Before **AutoRequest()** and **EasyRequest()** return control to the application, Intuition must be free to run and accept a response from the user. If the user presses the menu button, Intuition will wait for the program to reply to the IDCMP_MENUVERIFY event and a deadlock results.

Therefore, it is extremely important to use **ModifyIDCMP()** to turn off all verify messages before calling **AutoRequest()**, **EasyRequest()** or, directly or indirectly, AmigaDOS, since many error conditions in the DOS require user input in the form of an **EasyRequest()**. Indirect DOS calls include **OpenLibrary()**, **OpenDevice()**, and **OpenDiskFont()**.

Beginning with V36, all windows that have the IDCMP_MENUVERIFY bit set must respond to Intuition within a set time period, or the menu operation will time out and the menu action will be canceled. This prevents the deadlocks that were possible under previous versions of the operating system.

Menu Data Structures

The specifications for the menu structures are given below. **Menus** are the headers that show in the menu bar, and **MenuItems** are the items and sub-items that can be chosen by the user.

MENU STRUCTURE

Here is the specification for a **Menu** structure:

```
struct Menu
    {
    struct Menu *NextMenu;
    WORD LeftEdge, TopEdge;
    WORD Width, Height;
    UWORD Flags;
    BYTE *MenuName;
    struct MenuItem *FirstItem;
    WORD JazzX, JazzY, BeatX, BeatY;
    };
```

The variables in the **Menu** structure have the following meanings:

NextMenu
This variable points to the next **Menu** header in the list. The last **Menu** in the list should have a **NextMenu** value of NULL.

LeftEdge, TopEdge, Width, Height
These fields describe the select box of the header. Currently, any values supplied for **TopEdge** and **Height** are ignored by Intuition, which uses instead the screen's **TopBorder** for the **TopEdge** and the height of the screen's title bar for the **Height**.

LeftEdge is relative to the **LeftEdge** of the screen plus the screen's left border width, so if **LeftEdge** is 0, Intuition puts this header at the leftmost allowable position.

Flags

The flag space is shared by the program and Intuition. The flags are:

MENUENABLED

This flag is for Intuition's use and indicates whether or not this **Menu** is currently enabled. Set this flag before submitting the menu strip to Intuition. If this flag is not set, the menu header and all menu items below it will be disabled, and the user will be able to view, but not select any of the items. After submitting the strip to Intuition, the disabled or enabled status may be changed by calling **OnMenu()** or **OffMenu()**.

MIDRAWN

This flag indicates whether or not this menu's items are currently displayed to the user.

MenuName

This is a pointer to a NULL terminated character string that is printed on the screen's title bar starting at the **LeftEdge** of this menu's select box and at the **TopEdge** just below the screen title bar's top border. The text is rendered in the screen's font.

FirstItem

This points to the first **MenuItem** structure in the linked list of this menu's items.

JazzX, JazzY, BeatX, BeatY

For internal use only.

MENUITEM STRUCTURE

The **MenuItem** structure is used for both items and sub-items. There is no internal difference between items and sub-items, other than how they are linked into the menu strip. Items are linked directly to **Menu** structures through the **FirstItem** field, sub-items are linked to **MenuItem** structures through the **SubItem** field.

Here is the specification:

```
struct MenuItem
    {
    struct MenuItem *NextItem;
    WORD LeftEdge, TopEdge;
    WORD Width, Height;
    UWORD Flags;
    LONG MutualExclude;
    APTR ItemFill;
    APTR SelectFill;
    BYTE Command;
    struct MenuItem *SubItem;
    UWORD NextSelect;
    };
```

The fields have the following meanings:

NextItem

This field is a pointer to the next item in the list. The last item in the list should have a **NextItem** value of NULL.

LeftEdge, TopEdge, Width, Height

These fields describe the select box of the **MenuItem**. The **LeftEdge** is relative to the **LeftEdge** of the **Menu**. The **TopEdge** is relative to the topmost position Intuition allows. **TopEdge** is based on the way the user has the system configured--which font, which resolution, and so on. Use 0 for the topmost position.

Flags

The flag space is shared by the program and Intuition. See "MenuItem Flags" below for a description of the flag bits.

MutualExclude

Use these bits to describe which of the other items, if any, are mutually excluded by this one. This long word refers to the items in the same menu as this one. A maximum of 32 items may be described by this variable, and they must be the first 32 items in the menu. This does not mean that there cannot be more than 32 items in any given menu, just that only the first 32 can be mutually excluded.

ItemFill

This points to the data used in rendering this **MenuItem**. It can point to either an instance of an **IntuiText** structure with text for this **MenuItem** or an instance of an **Image** structure with image data. The program tells Intuition the type of data pointed to by this variable by setting or clearing the **MenuItem** flag bit ITEMTEXT. (See "MenuItem Flags" below for more details about ITEMTEXT.)

Note that the **IntuiText** or **Image** data need not be simple imagery, either of them may consist of multiple objects of the same type chained together as described in the chapter "Intuition Images, Line Drawing and Text". By chaining multiple **IntuiText** structures, the application may "fine tune" the positioning of text within each item. This is especially important for proportional fonts, where the width of the individual characters is not constant. This also allows items to have part of the text left justified and part right justified.

SelectFill

If HIGHIMAGE is set in the **Flags** variable as the **MenuItem** highlighting mode, Intuition substitutes this alternate image or text for the original rendering described by **ItemFill**. The type of this structure must be the same as **ItemFill**. **SelectFill** can point to either an **Image** or an **IntuiText**, where the type is determined by the setting of the ITEMTEXT flag.

Command

This variable is storage for a single alphanumeric character to be used as the command key substitute for this menu item. The command key sequence will be rendered in the menu display to the right of the item's select area, with a fancy, reverse-video A, followed by the command character. Case is ignored.

If the flag COMMSEQ is set, the user can hold down the right Amiga key on the keyboard (to mimic using the right mouse menu button) and press the indicated key as a shortcut for using the mouse to select this item.

SubItem

If this item has a sub-item list, this variable should point to the MenuItem structure describing the first sub-item in the list.

There Is No Such Thing as a Sub-sub-item. A sub-item cannot have a sub-item attached to it. If this **MenuItem** structure is not an item, this variable is ignored.

NextSelect

This field is filled in by Intuition when this **MenuItem** is selected by the user. After an item has been selected by the user, the program should process the request and then check the **NextSelect** field. If the **NextSelect** field is equal to MENUNULL, no other items were selected; otherwise, there is another item to process. See "Menu Numbers and Menu Selection Messages" above for more information about user selections.

MENUITEM FLAGS

Here are the flags that can be set by the application in the **Flags** field of the **MenuItem** structure:

CHECKIT

Set this flag to inform Intuition that this item is a checkmark item and should be preceded by a checkmark if the flag CHECKED is set.

CHECKED

For an item with the CHECKIT flag set, set this bit to specify that the checkmark is displayed. After the menu strip is submitted to Intuition, it will maintain the CHECKED bit based on effects from other items' mutual exclusions, or, for MENUTOGGLE items, from user accesses to this item.

ITEMTEXT

Set this flag if the representation of the item pointed to by the **ItemFill** field and, possibly, by **SelectFill** is text and points to an **IntuiText** structure. Clear this bit if the item is graphic and points to an **Image** structure.

COMMSEQ

If this flag is set, this item has an equivalent command key sequence set in the **Command** field of the **MenuItem** structure.

MENUTOGGLE

This flag is used in conjunction with the CHECKIT flag. If MENUTOGGLE is set, a checkmark that is turned on may be turned off by selecting the item. This allows the user to toggle between the checked and non-checked states by repeatedly selecting the item.

ITEMENABLED

This flag describes whether or not this item is currently enabled. If an item is not enabled, its image will be ghosted and the user will not be able to select it. If this item has sub-items, all of the sub-items are disabled when the item is disabled.

Set this flag before submitting the menu strip to Intuition. Once the menu strip has been submitted to Intuition, enable or disable items by calling **OnMenu()** or **OffMenu()**.

HIGHFLAGS

An item can be highlighted when the user positions the pointer over the item. These bits describe what type of highlighting will be used, if any. One of the following bits must be set, according to the type of highlighting desired:

HIGHCOMP

This complements all of the bits contained by this item's select box.

HIGHBOX

This draws a box outside this item's select box.

HIGHIMAGE

This displays alternate imagery referenced in **SelectFill**. For alternate text, make sure that ITEMTEXT is set, and that the **SelectFill** field points to an **IntuiText** structure. For alternate image, ITEMTEXT must be cleared, and the **SelectFill** field must point to an **Image** structure.

HIGHNONE

This specifies no highlighting.

The following two flags are used by Intuition:

ISDRAWN

Intuition sets this flag when this item's sub-items are currently displayed to the user and clears it when they are not.

HIGHITEM

Intuition sets this flag when this item is highlighted and clears it when the item is not highlighted.

A Menu Example

This example shows how to implement menus. The menu code is simply part of the processing for Intuition messages as shown in the IDCMP example in the "Intuition Input and Output Methods" chapter. The example implements extended selection for menus, adaptation to fonts of different sizes, mutual exclusion and checkmarks.

If possible, applications should use the menu layout routines available in the GadTools library, rather than doing the job themselves as this example does. See the "GadTools Library" chapter for more information.

```
;/* menulayout.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 menulayout.c
Blink FROM LIB:c.o,menulayout.o TO menulayout LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
** menulayout.c - Example showing how to do menu layout in general.  This example
** also illustrates handling menu events, including IDCMP_MENUHELP events.
**
** Note that handling arbitrary fonts is fairly complex.  Applications that require V37
** should use the simpler menu layout routines found in the GadTools library.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/gfxbase.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif
```

```
/* Our function prototypes */
BOOL processMenus(USHORT selection, BOOL done);
BOOL handleIDCMP(struct Window *win);
USHORT MaxLength(struct RastPort *textRPort, struct MenuItem *first_item, USHORT char_size);
VOID setITextAttr(struct IntuiText *first_IText, struct TextAttr *textAttr);
VOID adjustItems(struct RastPort *textRPort, struct MenuItem *first_item, struct TextAttr *textAttr,
                 USHORT char_size, USHORT height, USHORT level, USHORT left_edge);
BOOL adjustMenus(struct Menu *first_menu, struct TextAttr *textAttr);
LONG doWindow(void);

/* Settings Item IntuiText */
struct IntuiText SettText[] = {
        {0,1,JAM2,2,         1, NULL, "Sound...",        NULL },
        {0,1,JAM2,CHECKWIDTH,1, NULL, " Auto Save",      NULL },
        {0,1,JAM2,CHECKWIDTH,1, NULL, " Have Your Cake", NULL },
        {0,1,JAM2,CHECKWIDTH,1, NULL, " Eat It Too",     NULL }
    };

struct MenuItem SettItem[] = {
      { /* "Sound..." */
         &SettItem[1], 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
         (APTR)&SettText[0], NULL, NULL, NULL, MENUNULL },
        { /* "Auto Save" (toggle-select, initially selected) */
         &SettItem[2], 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT|MENUTOGGLE|CHECKED, 0,
         (APTR)&SettText[1], NULL, NULL, NULL, MENUNULL },
      { /* "Have Your Cake" (initially selected, excludes "Eat It Too") */
         &SettItem[3], 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT|CHECKED, 8,
         (APTR)&SettText[2], NULL, NULL, NULL, MENUNULL },
        { /* "Eat It Too" (excludes "Have Your Cake") */
         NULL, 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP|CHECKIT, 4,
         (APTR)&SettText[3], NULL, NULL, NULL, MENUNULL }
    };

/* Edit Menu Item IntuiText */
struct IntuiText EditText[] = {
        {0,1,JAM2,2,1, NULL, "Cut",       NULL },
        {0,1,JAM2,2,1, NULL, "Copy",      NULL },
        {0,1,JAM2,2,1, NULL, "Paste",     NULL },
        {0,1,JAM2,2,1, NULL, "Erase",     NULL },
        {0,1,JAM2,2,1, NULL, "Undo",      NULL }
    };

/* Edit Menu Items */
struct MenuItem EditItem[] = {
        { /* "Cut" (key-equivalent: 'X') */
          &EditItem[1], 0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&EditText[0], NULL, 'X', NULL, MENUNULL },
        { /* "Copy" (key-equivalent: 'C') */
          &EditItem[2], 0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&EditText[1], NULL, 'C', NULL, MENUNULL },
        { /* "Paste" (key-equivalent: 'V') */
          &EditItem[3], 0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&EditText[2], NULL, 'V', NULL, MENUNULL },
        { /* "Erase" (disabled) */
          &EditItem[4], 0, 0, 0, 0, ITEMTEXT|HIGHCOMP, 0,
          (APTR)&EditText[3], NULL, NULL, NULL, MENUNULL },
        { /* "Undo" MenuItem (key-equivalent: 'Z') */
          NULL, 0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&EditText[4], NULL, 'Z', NULL, MENUNULL }
    };

/* IntuiText for the Print Sub-Items */
struct IntuiText PrtText[] = {
        {0,1, JAM2,2,1, NULL, "NLQ",   NULL },
        {0,1, JAM2,2,1, NULL, "Draft", NULL }
    };

/* Print Sub-Items */
struct MenuItem PrtItem[] = {
        { /* "NLQ" */
          &PrtItem[1], 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&PrtText[0], NULL, NULL, NULL, MENUNULL },
        { /* "Draft" */
          NULL, 0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&PrtText[1], NULL, NULL, NULL, MENUNULL }
    };
```

```
/* Uses the >> character to indicate a sub-menu item.
** This is \273 Octal, 0xBB Hex or Alt-0 from the Keyboard.
**
** NOTE that standard menus place this character at the right margin of the menu box.
** This may be done by using a second IntuiText structure for the single character,
** linking this IntuiText to the first one, and positioning the IntuiText so that the
** character appears at the right margin.  GadTools library will provide the correct behavior.
*/

/* Project Menu Item IntuiText */
struct IntuiText ProjText[] = {
        {0,1, JAM2,2,1, NULL, "New",           NULL },
        {0,1, JAM2,2,1, NULL, "Open...",       NULL },
        {0,1, JAM2,2,1, NULL, "Save",          NULL },
        {0,1, JAM2,2,1, NULL, "Save As...",    NULL },
        {0,1, JAM2,2,1, NULL, "Print     \273", NULL },
        {0,1, JAM2,2,1, NULL, "About...",      NULL },
        {0,1, JAM2,2,1, NULL, "Quit",          NULL }
    };


/* Project Menu Items */
struct MenuItem ProjItem[] = {
        { /* "New" (key-equivalent: 'N' */
          &ProjItem[1],0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[0], NULL, 'N', NULL, MENUNULL },
        { /* "Open..." (key-equivalent: 'O') */
          &ProjItem[2],0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[1], NULL, 'O', NULL, MENUNULL },
        { /* "Save" (key-equivalent: 'S') */
          &ProjItem[3],0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[2], NULL, 'S', NULL, MENUNULL },
        { /* "Save As..." (key-equivalent: 'A') */
          &ProjItem[4],0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[3], NULL, 'A', NULL, MENUNULL },
        { /* "Print" (has sub-menu) */
          &ProjItem[5],0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[4], NULL, NULL, &PrtItem[0], MENUNULL },
        { /* "About..." */
          &ProjItem[6],0, 0, 0, 0, ITEMTEXT|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[5], NULL, NULL, NULL, MENUNULL },
        { /* "Quit" (key-equivalent: 'Q' */
          NULL, 0, 0, 0, 0, ITEMTEXT|COMMSEQ|ITEMENABLED|HIGHCOMP, 0,
          (APTR)&ProjText[6], NULL, 'Q', NULL, MENUNULL }
    };

/* Menu Titles */
struct Menu Menus[] = {
        {&Menus[1],   0, 0, 63, 0, MENUENABLED, "Project",    &ProjItem[0]},
        {&Menus[2],  70, 0, 39, 0, MENUENABLED, "Edit",       &EditItem[0]},
        {NULL,      120, 0, 88, 0, MENUENABLED, "Settings",   &SettItem[0]},
    };

/* A pointer to the first menu for easy reference */
struct Menu *FirstMenu = &Menus[0];

/* Window Text for Explanation of Program */
struct IntuiText WinText[] = {
        {0, 0, JAM2, 0, 0, NULL, "How to do a Menu", NULL},
        {0, 0, JAM2, 0, 0, NULL, "(with Style)", &WinText[0]}
    };

/* Globals */
struct Library  *IntuitionBase = NULL;
struct Library  *GfxBase = NULL;

/* open all of the required libraries.  Note that we require
** Intuition V37, as the routine uses OpenWindowTags().
*/
VOID main(int argc, char **argv)
{
LONG returnValue;

/* This gets set to RETURN_OK if everything goes well. */
returnValue = RETURN_FAIL;
```

```
/* Open the Intuition Library */
IntuitionBase = OpenLibrary("intuition.library", 37);
if (IntuitionBase)
    {
    /* Open the Graphics Library */
    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33);
    if (GfxBase)
        {
        returnValue = doWindow();

        CloseLibrary(GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
exit(returnValue);
}

/* Open a window with some properly positioned text.  Layout and set
** the menus, then process any events received.  Cleanup when done.
*/

LONG doWindow()
{
struct Window *window;
struct Screen *screen;
struct DrawInfo *drawinfo;
ULONG  signalmask, signals;
ULONG  win_width, alt_width, win_height;
LONG   returnValue = RETURN_FAIL;
BOOL   done = FALSE;

if (screen = LockPubScreen(NULL))
    {
    if (drawinfo = GetScreenDrawInfo(screen))
        {
        /* get the colors for the window text */
        WinText[0].FrontPen = WinText[1].FrontPen = drawinfo->dri_Pens[TEXTPEN];
        WinText[0].BackPen  = WinText[1].BackPen  = drawinfo->dri_Pens[BACKGROUNDPEN];

        /* use the screen's font for the text */
        WinText[0].ITextFont = WinText[1].ITextFont = screen->Font;

        /* calculate window size */
        win_width  = 100 + IntuiTextLength(&(WinText[0]));
        alt_width  = 100 + IntuiTextLength(&(WinText[1]));
        if (win_width < alt_width)
            win_width  = alt_width;
        win_height = 1 + screen->WBorTop + screen->WBorBottom +
                     (screen->Font->ta_YSize * 5);

        /* calculate the correct positions for the text in the window */
        WinText[0].LeftEdge = (win_width - IntuiTextLength(&(WinText[0]))) >> 1;
        WinText[0].TopEdge  = 1 + screen->WBorTop + (2 * screen->Font->ta_YSize);
        WinText[1].LeftEdge = (win_width - IntuiTextLength(&(WinText[1]))) >> 1;
        WinText[1].TopEdge  = WinText[0].TopEdge + screen->Font->ta_YSize;

        /* Open the window */
        window = OpenWindowTags(NULL,
            WA_PubScreen, screen,
            WA_IDCMP,     IDCMP_MENUPICK | IDCMP_CLOSEWINDOW | IDCMP_MENUHELP,
            WA_Flags,     WFLG_DRAGBAR | WFLG_DEPTHGADGET | WFLG_CLOSEGADGET |
                              WFLG_ACTIVATE | WFLG_NOCAREREFRESH,
            WA_Left,      10,             WA_Top,       screen->BarHeight + 1,
            WA_Width,     win_width,      WA_Height,    win_height,
            WA_Title,     "Menu Example", WA_MenuHelp,  TRUE,
            TAG_END);


        if (window)
            {
            returnValue = RETURN_OK;  /* program initialized ok */

            /* Give a brief explanation of the program */
            PrintIText(window->RPort,&WinText[1],0,0);
```

```
            /* Adjust the menu to conform to the font (TextAttr) */
            adjustMenus(FirstMenu, window->WScreen->Font);

            /* attach the menu to the window */
            SetMenuStrip(window, FirstMenu);

            /* Set up the signals that you want to hear about ... */
            signalmask = 1L << window->UserPort->mp_SigBit;

            /* And wait to hear from your signals */
            while (!done)
                {
                signals = Wait(signalmask);
                if (signals & signalmask) done = handleIDCMP(window);
                };

            /* clean up everything used here */
            ClearMenuStrip(window);
            CloseWindow(window);
            }
        FreeScreenDrawInfo(screen,drawinfo);
        }
    UnlockPubScreen(NULL,screen);
    }
return(returnValue);
}

/* print out what menu was selected.  Properly handle the IDCMP_MENUHELP
** events.  Set done to TRUE if quit is selected.
*/
BOOL processMenus(USHORT selection, BOOL done)
{
USHORT flags;
USHORT menuNum, itemNum, subNum;

menuNum = MENUNUM(selection);
itemNum = ITEMNUM(selection);
subNum  = SUBNUM(selection);

/* when processing IDCMP_MENUHELP, you are not guaranteed
** to get a menu item.
*/
if (itemNum != NOITEM)
    {
    flags = ((struct MenuItem *)ItemAddress(FirstMenu,(LONG)selection))->Flags;
    if (flags & CHECKED)
        printf("(Checked) ");
    }

switch (menuNum)
    {
    case 0:      /* Project Menu */
        switch (itemNum)
            {
            case NOITEM: printf("Project Menu\n");        break;
            case 0:      printf("New\n");                 break;
            case 1:      printf("Open\n");                break;
            case 2:      printf("Save\n");                break;
            case 3:      printf("Save As\n");             break;
            case 4:      printf("Print ");
                switch (subNum)
                    {
                    case NOSUB: printf("Item\n");  break;
                    case 0:     printf("NLQ\n");   break;
                    case 1:     printf("Draft\n"); break;
                    }
                break;
            case 5:      printf("About\n");               break;
            case 6:      printf("Quit\n");   done = TRUE; break;
            }
        break;
```

```
    case 1:      /* Edit Menu */
        switch (itemNum) {
            case NOITEM: printf("Edit Menu\n"); break;
            case 0:      printf("Cut\n");       break;
            case 1:      printf("Copy\n");      break;
            case 2:      printf("Paste\n");     break;
            case 3:      printf("Erase\n");     break;
            case 4:      printf("Undo\n");      break;
            }
        break;
    case 2:      /* Settings Menu */
        switch (itemNum) {
            case NOITEM: printf("Settings Menu\n"); break;
            case 0:      printf("Sound\n");          break;
            case 1:      printf("Auto Save\n");      break;
            case 2:      printf("Have Your Cake\n"); break;
            case 3:      printf("Eat It Too\n");     break;
            }
        break;
    case NOMENU: /* No menu selected, can happen with IDCMP_MENUHELP */
        printf("no menu\n");
        break;
    }
return(done);
}

/* Handle the IDCMP messages.  Set done to TRUE if quit or closewindow is selected. */
BOOL handleIDCMP(struct Window *win)
{
BOOL done;
USHORT code, selection;
struct IntuiMessage *message = NULL;
ULONG  class;

done = FALSE;

/* Examine pending messages */
while (message = (struct IntuiMessage *)GetMsg(win->UserPort)) {
    class = message->Class;
    code = message->Code;

    /* When we're through with a message, reply */
    ReplyMsg((struct Message *)message);

    /* See what events occurred */
    switch (class) {
        case IDCMP_CLOSEWINDOW:
            done = TRUE;
            break;
        case IDCMP_MENUHELP:
            /*
          ** The routine that handles the menus for IDCMP_MENUHELP must be very careful
          ** it can receive menu information that is impossible under IDCMP_MENUPICK.
          ** For instance, the code value on a IDCMP_MENUHELP may have a valid number
          ** for the menu, then NOITEM and NOSUB. IDCMP_MENUPICK would get MENUNULL
          ** in this case.  IDCMP_MENUHELP never come as multi-select items, and the
          ** event terminates the menu processing session.
            **
            ** Note that I do not keep the return value from the processMenus() routine here--the
          ** application should not quit if the user selects "help" over the quit menu item.
            */
            printf("IDCMP_MENUHELP: Help on ");
            processMenus(code,done);
            break;
        case IDCMP_MENUPICK:
            for ( selection = code; selection != MENUNULL;
                  selection = (ItemAddress(FirstMenu, (LONG)selection))->NextSelect)
                {
                printf("IDCMP_MENUPICK: Selected ");
                done = processMenus(selection,done);
                }
            break;
        }
    }
return(done);
}
```

```
/* Steps thru each item to determine the maximum width of the strip */
USHORT MaxLength(struct RastPort *textRPort, struct MenuItem *first_item, USHORT char_size)
{
USHORT maxLength;
USHORT total_textlen;
struct MenuItem  *cur_item;
struct IntuiText *itext;
USHORT extra_width;
USHORT maxCommCharWidth;
USHORT commCharWidth;

extra_width = char_size;  /* used as padding for each item. */

/* Find the maximum length of a command character, if any.
** If found, it will be added to the extra_width field.
*/
maxCommCharWidth = 0;
for (cur_item = first_item; cur_item != NULL; cur_item = cur_item->NextItem)
    {
    if (cur_item->Flags & COMMSEQ)
        {
        commCharWidth = TextLength(textRPort,&(cur_item->Command),1);
        if (commCharWidth > maxCommCharWidth)
            maxCommCharWidth = commCharWidth;
        }
    }

/* if we found a command sequence, add it to the extra required space.  Add
** space for the Amiga key glyph plus space for the command character.  Note
** this only works for HIRES screens, for LORES, use LOWCOMMWIDTH.
*/
if (maxCommCharWidth > 0)
    extra_width += maxCommCharWidth + COMMWIDTH;

/* Find the maximum length of the menu items, given the extra width calculated above. */
maxLength = 0;
for (cur_item = first_item; cur_item != NULL; cur_item = cur_item->NextItem)
    {
    itext = (struct IntuiText *)cur_item->ItemFill;
    total_textlen = extra_width + itext->LeftEdge +
          TextLength(textRPort, itext->IText, strlen(itext->IText));

    /* returns the greater of the two */
    if (total_textlen > maxLength)
        maxLength = total_textlen;
    }
return(maxLength);
}

/* Set all IntuiText in a chain (they are linked through the NextText ** field) to the same font. */
VOID setITextAttr(struct IntuiText *first_IText, struct TextAttr *textAttr)
{
struct IntuiText *cur_IText;

for (cur_IText = first_IText; cur_IText != NULL; cur_IText = cur_IText->NextText)
    cur_IText->ITextFont = textAttr;
}


/* Adjust the MenuItems and SubItems */
VOID adjustItems(struct RastPort *textRPort, struct MenuItem *first_item,
                 struct TextAttr *textAttr, USHORT char_size, USHORT height,
                 USHORT level, USHORT left_edge)
{
register USHORT item_num;
struct   MenuItem *cur_item;
USHORT   strip_width, subitem_edge;

if (first_item == NULL)
    return;

/* The width of this strip is the maximum length of its members. */
strip_width = MaxLength(textRPort, first_item, char_size);
```

```
/* Position the items. */
for (cur_item = first_item, item_num = 0; cur_item != NULL; cur_item = cur_item->NextItem, item_num++)
    {
    cur_item->TopEdge  = (item_num * height) - level;
    cur_item->LeftEdge = left_edge;
    cur_item->Width    = strip_width;
    cur_item->Height   = height;

    /* place the sub_item 3/4 of the way over on the item. */
    subitem_edge = strip_width - (strip_width >> 2);

    setITextAttr((struct IntuiText *)cur_item->ItemFill, textAttr);
    adjustItems(textRPort,cur_item->SubItem,textAttr,char_size,height,1,subitem_edge);
    }
}


/* The following routines adjust an entire menu system to conform to the specified fonts' width and
** height.  Allows for Proportional Fonts. This is necessary for a clean look regardless of what the
** users preference in Fonts may be.  Using these routines, you don't need to specify TopEdge,
** LeftEdge, Width or Height in the MenuItem structures.
**
** NOTE that this routine does not work for menus with images, but assumes that all menu items are
** rendered with IntuiText.
**
** This set of routines does NOT check/correct if the menu runs off
** the screen due to large fonts, too many items, lo-res screen.
*/
BOOL adjustMenus(struct Menu *first_menu, struct TextAttr *textAttr)
{
struct RastPort   textrp = {0};            /* Temporary RastPort */
struct Menu      *cur_menu;
struct TextFont  *font;                    /* Font to use */
USHORT            start, char_size, height;
BOOL              returnValue = FALSE;

/* open the font */
if (font = OpenFont(textAttr))
    {
    SetFont(&textrp, font);        /* Put font into temporary RastPort */

    char_size = TextLength(&textrp, "n", 1);    /* Get the Width of the Font */

    /* To prevent crowding of the Amiga key when using COMMSEQ, don't allow the items to be less
    ** than 8 pixels high.  Also, add an extra pixel for inter-line spacing.
    */
    if (font->tf_YSize > 8)
        height = 1 + font->tf_YSize;
    else
        height = 1 + 8;

    start = 2;      /* Set Starting Pixel */

    /* Step thru the menu structure and adjust it */
    for (cur_menu = first_menu; cur_menu != NULL; cur_menu = cur_menu->NextMenu)
        {
        cur_menu->LeftEdge = start;
        cur_menu->Width = char_size +
            TextLength(&textrp, cur_menu->MenuName, strlen(cur_menu->MenuName));
        adjustItems(&textrp, cur_menu->FirstItem, textAttr, char_size, height, 0, 0);
        start += cur_menu->Width + char_size + char_size;
        }
    CloseFont(font);             /* Close the Font */
    returnValue = TRUE;
    }
return(returnValue);
}
```

Other Menu Macros

The **MENUNEM()**, **ITEMNUM()** and **SUBNUM()** macros let an application break a menu number down into its component parts--the specific menu number, the item number and the sub-item number. (See the section on "Menu Numbers" earlier in this chapter for details.) Intuition also supplies macros that allow an application to construct a menu number given its components:

SHIFTMENU(n)
Create a properly masked and shifted specific menu number.

SHIFTITEM(n)
Create a properly masked and shifted item number.

SHIFTSUB(n)
Create a properly masked and shifted sub-item number.

FULLMENUNUM(menu, item, sub)
Create a complete composite menu number from its components.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of Intuition menus. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 6-2: Functions for Intuition Menus

Function	Description
SetMenuStrip()	Set a menu for an open window.
ClearMenuStrip()	Clear the menu of an open window.
ResetMenuStrip()	Set a pre-calculated menu for an open window.
ItemAddress()	Find the address of a menu item from its position.
OffMenu()	Disable a menu in a menu strip.
OnMenu()	Enable a menu in a menu strip.

Chapter 7
INTUITION REQUESTERS AND ALERTS

This chapter explains how to create requesters, the information exchange boxes that both the system and applications can use for confirming actions, getting command options and similar operations. These boxes are called requesters because they generally request information from the user.

Alerts provide a function similar to requesters but are reserved for emergency messages. Alerts are discussed later in this chapter.

Types Of Requesters

There are at least three kinds of display objects in Amiga terminology called requesters: *true requesters*, *system requesters* and *ASL requesters.*

True requesters are general purpose display areas that can be thought of as temporary sub-windows. They display information to the user and allow the user to make a selection. True requesters always open within an existing window and are constrained to the boundaries of that window (often referred to as the *parent* window). If a requester extends beyond the edge of its parent window, either its position is adjusted or its graphics are clipped. True requesters always block input to their parent window as long as they are present.

System requesters are typically used for warnings or to confirm an action the user has just initiated. System requesters differ from true requesters in that they cannot block input to the parent window. In fact, system requesters do not open in a parent window at all, but instead open their own separate window in the screen. Since these requesters are so different from true requesters, they will be discussed separately later in the chapter. See the sections on ''Easy Requesters'' and ''System Requests'' for more information.

The third type of requester, the ASL requester, is a special purpose requester available only in Release 2 and later versions of the OS. ASL requesters provide an easy, standard way to get a filename from the user for load and save operations. They can also be used to get a font selection from the user. Since selecting a file or font name is one of the most common uses for a requester, it has been incorporated into Release 2 as a standard feature. For the details about ASL file and font requesters, see Chapter 16, ''ASL Library''.

True Requesters

The primary function of a requester is to display information to the user from which the user is to make a selection. Conceptually, requesters are similar to menus since both menus and requesters offer options to the user. Requesters, however, go beyond menus because they can have customized imagery, can be placed anywhere in a window, can be activated by the application and may have any type of gadget attached.

For instance, to select a color for a given operation using a menu could be awkward, especially in an application that supports a large number of colors. In that case a requester could be used instead (see figure).

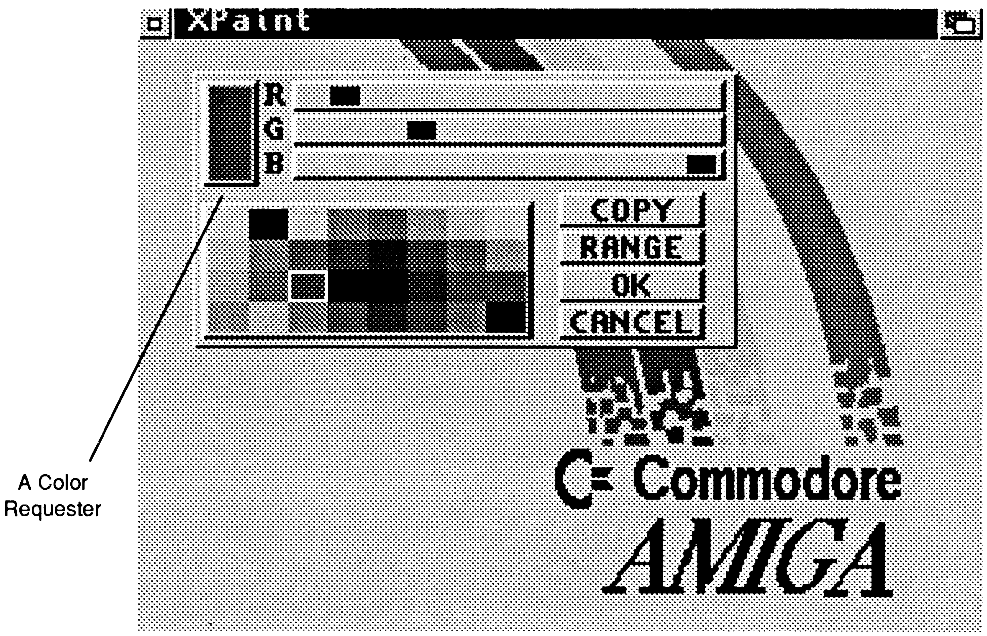


Figure 7-1: Requester Deluxe

The ability of a true requester to block input to its parent window is important in understanding how requesters are used. When input is blocked by a true requester (also known as a modal requester), the user must take some action before the program will proceed further, such as making a selection, correcting an error condition, or acknowledging a warning. These are situations where a true (modal) requester is appropriate, however, keep in mind that your application should try to be as user-responsive as possible. Putting up a requester merely because you are in a phase of the program where it would be difficult to deal with user input is bad style. Modal requesters should be used only when the program *requires* user interaction before proceeding.

True requesters can be created in a window in two different ways.

- An application can display a requester at any time by calling the **Request()** function.
- The application can declare a requester as the window's *double menu requester*, which the user can bring up with a double-click of the menu button (this method is rarely used).

CREATING APPLICATION REQUESTERS

To create a requester, the application first allocates memory for or declares an instance of the **Requester** structure as defined in *<intuition/intuition.h>*. Once the **Requester** structure is set up, it is initialized with the **InitRequester()** function.

```
void InitRequester( struct Requester *requester)
```

This function simply clears the **Requester** structure. The application should do further initialization depending on its needs. See the section on the "Requester Structure" below for an explanation of all the **Requester** fields and how to set them.

A true (modal) requester is attached to its parent window and displayed with the **Request()** function.

```
BOOL Request(struct Requester *requester, struct Window *window)
```

This function returns TRUE if the requester opens successfully or FALSE if the requester cannot be opened. If the requester opens successfully, menu and gadget input in the parent window is blocked as long as the requester is displayed. The application should process input events from the requester, which are sent to the parent window's **Window.UserPort**, until the requester is satisfied.

To remove a requester from its parent window and update the display, use **EndRequest()**.

```
void EndRequest( struct Requester *requester, struct Window *window );
```

This removes only the one requester specified. It is possible to set up a requester with a special gadget that, if selected, will automatically close the requester. In that case, **EndRequest()** need not be called. If the program needs to cancel the request early, or cancel it only after some specific manipulation of the gadgets, **EndRequest()** should be used.

The application should always provide a safe way for the user to back out of a requester without taking any action that affects the user's work. Providing an escape hatch is important, for instance, a requester with the message "Overwrite File?" should allow the user to cancel the operation without losing the old data.

REQUESTER I/O

So long as a requester is active in a window, the only gadgets that can be used are those that are in the requester, plus all of the window's system gadgets except for the close gadget (i.e., the drag bar, size gadget, depth gadget, and zoom gadget). A requester also makes the menus of the parent window inaccessible. Additionally, mouse button and keyboard events will be blocked (unless the requester's NOISYREQ flag is set; see "Requester Structure" below). Mouse movement events, if enabled in the parent window (with WFLG_REPORTMOUSE), are not blocked.

Requesters do not have their own IDCMP message ports. Instead, events for a requester are sent to the IDCMP port of the requester's parent window (**Window.UserPort**). Since the window's menus and application gadgets are inaccessible, no IDCMP events will be sent for them.

Even though the window containing the requester is blocked for input, the user can work in another application or even in a different window of the same application without satisfying the requester. Only input to the parent window is blocked by a requester.

Output is not blocked by a requester so nothing prevents the application from writing to the window. Be aware, however, that the requester obscures part of the display and cannot be moved within the window so this may limit the usefulness of any output you send to the parent window. There are several ways to monitor the comings and goings of requesters that allow the program to know if requesters are currently displayed in a given window. See "IDCMP Requester Features" below.

The information displayed in a requester is placed in its own layer, so it does not overwrite the information in the window. Output to the window while the requester is displayed will not change the requester's display, it will go into the window's layer. The requester's layer is clipped to the window's boundaries, so the data in the requester is only visible if the window is large enough to allow for the complete display of that data.

The requester will remain in the window and input will remain blocked until the user satisfies the request or the application removes the requester. Applications can set up some or all of the gadgets in the requester to automatically terminate the requester when the gadget is selected. This allows the requester to be removed from the window by user action. The application may also remove requesters from the window based on some event internal to the program.

Multiple requesters may be nested in a single window. Such requesters must be satisfied in the reverse order in which they were posted; the last requester to be displayed must be satisfied first. Input will not be restored to a previous requester until all later requesters are satisfied.

Note that the application may not bring up a limitless number of requesters in a window. Each requester creates a new layer for rendering in its window and the system currently has a limit of ten layers per window. Normal windows use one layer for the window rendering, GimmeZeroZero windows use a second layer for the border rendering. This leaves a maximum of eight or nine simultaneous requesters open in a window at any given time.

If the requester is being brought up only to display an error message, the application may want to use a less intrusive method of bringing the error to the user's attention than a requester. Requesters interrupt the flow of the user's work, and force them to respond before continuing.

As an alternative to bringing up an error requester, the application could flash the screen instead with Intuition's **DisplayBeep()** function. This allows the application to notify the user of an error that is not serious enough to warrant a requester and to which the user does not really need to respond. For more information, see the description of **DisplayBeep()** in the "Intuition Screens" chapter.

RENDERING REQUESTERS

The application may choose to use Intuition's rendering facilities to display the requester, or it may define its own custom bitmap. The **Requester** structure is initialized differently according to the rendering method chosen.

To use Intuition's rendering facilities, you supply a list of one or more display objects with the **Requester** structure and submit the **Requester** to Intuition, allowing it to draw the objects. These objects can include independent lists of **Borders**, **IntuiText**, **Images** and **Gadgets**. Note that the abilty to provide a list of **Image** structures is new in V36, and the USEREQIMAGE flag must be set for them to be rendered. For more about Intuition rendering see the chapter on "Intuition Images, Line Drawing and Text".

The gadgets in a requester also have their own borders, images and text to add to the display imagery. Intuition will allocate the buffers, construct a bitmap that lasts for the duration of the display, and render the requester into the window. This rendering is all done over a solid color, filled background specified by the **BackFill** pen in the **Requester** structure. The backfill may be disabled by setting the NOREQBACKFILL flag (this also a new feature of V36).

On the other hand, a custom requester may be designed with pre-defined, bitmap imagery for the entire object. The image bitmap is submitted to Intuition through the **ImageBMap** field of the **Requester** structure. The bitmap should be designed to reduce user confusion; gadgets should line up with their images, and the designer should attempt to use glyphs (symbols) familiar to the user.

To provide imagery for the requester, applications should always try to use data structures attached to the **Requester** structure as described above. Although, rendering directly into the requester layer's **RastPort** is tolerated, it must be done with great care.

First, a requester is allowed to have gadgets that automatically close the requester when they are selected (GACT_ENDGADGET). If such a gadget is selected, the requester, its layer, and its layer's **RastPort** will be deleted asynchronously to your application. If your application is trying to render directly into the requester at that time, the results are unpredictable. Therefore, do not put GACT_ENDGADGET gadgets into a requester if you plan on rendering directly into its **RastPort**.

Second, recall that requesters are clipped to the inside of the window (not including the borders). If the window can be sized smaller such that the requester would be entirely clipped, the requester's layer may be deleted by Intuition. If your window's minimum size and the requester size and position are such that the requester can be completely clipped, then reading **Requester.ReqLayer** is unsafe without additional protection. It would be correct to **LockLayerInfo()** the screen's **Layer_Info**, then check for the existence of the requester's **ReqLayer**, then render, then unlock.

For reasons such as these, direct rendering is discouraged.

REQUESTER REFRESH TYPE

A requester appears in a **Layer**. By default, the requester layer is of type LAYERSMART, or, in window terminology, WFLG_SMART_REFRESH; so rendering is preserved in the requester when the window is moved or revealed.

Requesters may also be simple refresh. This is the recommended type. If possible, make the requester a simple refresh layer requester by specifying the SIMPLEREQ flag.

For all refresh types, Intuition will keep the gadget, border, image and bitmap imagery properly refreshed.

REQUESTER DISPLAY POSITION

The location of true requesters may be specified in one of three ways. The requester may either be a constant location, which is an offset from the top left corner of the window; a location relative to the current location of the pointer; or a location relative to the center of the window.

To display the requester as an offset from the upper left corner of the window, initialize the **TopEdge** and **LeftEdge** variables and clear the POINTREL flag. This will create a requester with a fixed position relative to the upper left corner for both normal requesters and double menu requesters.

Displaying the requester relative to the pointer can get the user's attention immediately and closely associates the requester with whatever the user was doing just before the requester was displayed in the window. However, only double menu requesters may be positioned relative to the pointer position. See below for more information on double menu requesters.

Requesters that are not double menu requesters may be positioned relative to the center of the window on systems running Release 2 or a later version of the OS. This is done by setting the POINTREL flag and filling in the relative top and left of the gadget. Setting **RelTop** and **RelLeft** to zero will center the requester in the window. Positive values of **RelTop** and **RelLeft** will move the requester down and to the right, negative values will move it up and to the left.

GADGETS IN REQUESTERS

Each requester gadget must have the GTYP_REQGADGET flag set in the **GadgetType** field of its **Gadget** structure. This informs Intuition that this gadget is to be rendered in a requester rather than a window.

Requesters can have gadgets in them that *automatically* satisfy the request and end the requester. When one of these gadgets is selected, Intuition will remove the requester from the window. This is equivalent to the application calling **EndRequest**(), and, if the request is terminated by selection of such a gadget, the application should not call **EndRequest**() for that requester.

Set the GACT_ENDGADGET flag in the **Activation** field of the **Gadget** structure to create a gadget that automatically terminates the requester. Every time one of the requester's gadgets is selected, Intuition examines the GACT_ENDGADGET flag. If GACT_ENDGADGET is set, the requester is removed from the display and unlinked from the window's active requester list.

Requesters rendered via Intuition and those that use a custom bitmap differ in how their gadgets are rendered. For requesters rendered via Intuition, the application supplies a regular gadget list just as it would for application gadgets in a window.

In custom bitmap requesters, however, any gadget imagery is part of the bitmap supplied for the requester. Therefore the list of gadgets supplied for custom bitmap requesters should not provide gadget imagery but rather it should define only the select boxes, highlighting, and gadget types for the gadgets.

The **Gadget** structures used with a custom bitmap requester should have their **GadgetRender**, **SelectRender** and **GadgetText** fields set to NULL as these will be ignored. Other gadget information--select box dimensions, highlighting, and gadget type--is still relevant. The select box information is especially important since the select box must have a well defined correspondence with the custom bitmap imagery supplied. The basic idea is to make sure that the user understands the requester imagery and gadgets.

USING A REQUESTER TO BLOCK WINDOW INPUT

There may be times when an application needs to block user input without a visible requester. In some cases, the application needs to be busy for a while. Other times, an application wants the blocking properties of a requester, but prefers to use a window instead of a true requester. In this case, the application can create a requester with no imagery, attaching it to the parent window to block input. A new window may then be opened to act as the requester.

Some of the advantages of using a window as a requester instead of a real requester include:

- A window can be resized, and moves independently of the parent window.
- It is legal to render directly into a window.
- The window can have its own menus since only the parent window's menus are disabled (this is only occasionally useful).
- Certain code or a library you are using may not work in requesters (GadTools library is an example of this).

Of course, using a true requester instead of a window has the advantage that the requester automatically moves and depth-arranges along with the parent window.

A Requester Example

To use a window as a requester, first bring up a zero-sized requester attached to the main window (this provides the blocking feature). Then, bring up your second window, or bring up a busy pointer. The following example illustrates bringing up a busy pointer.

```
;/* blockinput.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 blockinput.c
Blink FROM LIB:c.o,blockinput.o TO blockinput LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** blockinput.c -- program to demonstrate how to block the input from a
** window using a minimal requester, and how to put up a busy pointer.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* our function prototypes */
BOOL beginWait(struct Window *win, struct Requester *waitRequest);
VOID endWait(struct Window *win, struct Requester *waitRequest);
VOID processIDCMP(struct Window *win);

struct Library *IntuitionBase;

/* data for a busy pointer.
** this data must be in chip memory!!!
*/
UWORD __chip waitPointer[] =
    {
    0x0000, 0x0000,     /* reserved, must be NULL */

    0x0400, 0x07C0,
    0x0000, 0x07C0,
    0x0100, 0x0380,
    0x0000, 0x07E0,
    0x07C0, 0x1FF8,
    0x1FF0, 0x3FEC,
    0x3FF8, 0x7FDE,
    0x3FF8, 0x7FBE,
    0x7FFC, 0xFF7F,
```

```
    0x7EFC, 0xFFFF,
    0x7FFC, 0xFFFF,
    0x3FF8, 0x7FFE,
    0x3FF8, 0x7FFE,
    0x1FF0, 0x3FFC,
    0x07C0, 0x1FF8,
    0x0000, 0x07E0,

    0x0000, 0x0000,     /* reserved, must be NULL */
    };



/*
** main()
**
** Open a window and display a busy-pointer for a short time then wait for
** the user to hit the close gadget (in processIDCMP()).  Normally, the
** application would bracket sections of code where it wishes to block window
** input with the beginWait() and endWait() functions.
*/
VOID main (int argc, char **argv)
{
struct Window *win;

if (IntuitionBase = OpenLibrary("intuition.library",37))
    {
    if (win = OpenWindowTags(NULL,
                        WA_IDCMP, IDCMP_CLOSEWINDOW|IDCMP_INTUITICKS,
                        WA_Activate, TRUE,
                        WA_Width,  320,
                        WA_Height, 100,
                        WA_CloseGadget, TRUE,
                        WA_DragBar, TRUE,
                        WA_DepthGadget, TRUE,
                        WA_SizeGadget, TRUE,
                        WA_MaxWidth, ~0,
                        WA_MaxHeight, ~0,
                        TAG_END))
        {
        processIDCMP(win);
        CloseWindow(win);
        }
    CloseLibrary(IntuitionBase);
    }
}



/*
** beginWait()
**
** Clear the requester with InitRequester.  This makes a requester of
** width = 0, height = 0, left = 0, top = 0; in fact, everything is zero.
** This requester will simply block input to the window until
** EndRequest is called.
**
** The pointer is set to a reasonable 4-color busy pointer, with proper offsets.
*/
BOOL beginWait(struct Window *win, struct Requester *waitRequest)
{
extern UWORD __chip waitPointer[];

InitRequester(waitRequest);
if (Request(waitRequest, win))
    {
    SetPointer(win, waitPointer, 16, 16, -6, 0);
    SetWindowTitles(win,"Busy - Input Blocked",(UBYTE *)~0);
    return(TRUE);
    }
else
    return(FALSE);
}
```

```
/*
** endWait()
**
** Routine to reset the pointer to the system default, and remove the
** requester installed with beginWait().
*/
VOID endWait(struct Window *win, struct Requester *waitRequest)
{
ClearPointer(win);
EndRequest(waitRequest, win);
SetWindowTitles(win,"Not Busy",(UBYTE *)~0);
}


/*
** processIDCMP()
**
** Wait for the user to close the window.
*/
VOID processIDCMP(struct Window *win)
{
WORD done;
struct IntuiMessage *msg;
ULONG class;
struct Requester myreq;
UWORD tick_count;

done = FALSE;

/* Put up a requester with no imagery (size zero). */
if (beginWait(win,&myreq))
    {
    /*
    ** Insert code here for a window to act as the requester.
    */

    /* We'll count down INTUITICKS, which come about ten times
    ** a second.  We'll keep the busy state for about three seconds.
    */
    tick_count = 30;
    }

while (!done)
    {
    Wait(1L << win->UserPort->mp_SigBit);

    while (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        class = msg->Class;
        ReplyMsg((struct Message *)msg);

        switch (class)
            {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;

            case IDCMP_INTUITICKS:
                if (tick_count > 0)
                    {
                    if (--tick_count == 0)  endWait(win,&myreq);
                    }
                break;
            }
        }
    }
}
```

DOUBLE MENU REQUESTERS

A double menu requester is exactly like other requesters with one exception: it is displayed only when the user double clicks the mouse menu button. Double menu requesters block input in exactly the same manner as other true requesters. A double menu requester is attached to a window by calling **SetDMRequest()**.

```
BOOL SetDMRequest( struct Window *window, struct Requester *requester );
```

This call does not display the requester, it simply prepares it for display. The requester will be brought up when the user double clicks the mouse menu button. The parent window will receive IDCMP_REQSET and IDCMP_REQCLEAR messages when the requester is added and removed.

To prevent the user from bringing up a double menu requester, unlink it from the window by calling **ClearDMRequest()**. If a double menu request is set for a window, **ClearDMRequest()** should be called to remove the requester before that window is closed.

```
BOOL ClearDMRequest( struct Window *window );
```

This function unlinks the requester from the window and disables the ability of the user to bring it up. **ClearDMRequest()** will fail if the double menu request is currently being displayed.

Double menu requesters can be positioned relative to the current mouse pointer position. For a mouse relative requester, specify POINTREL in the **Flags** field and initialize the **RelLeft** and **RelTop** variables. **RelLeft** and **RelTop** describe the offset of the upper, left corner of the requester from the pointer position at the time the requester is displayed. These values can be either negative or positive.

The values of **RelLeft** and **RelTop** are only advisory; the actual position will be restricted such that the requester is entirely contained within the borders of its parent window, if possible. The actual top and left positions are stored in the **TopEdge** and **LeftEdge** variables.

Positioning relative to the mouse pointer is possible only with double menu requesters. Setting POINTREL in a requester which is not a double menu requester will position the requester relative to the center of the window.

IDCMP REQUESTER FEATURES

Intuition can notify your application about user activity in the requester by sending a message to the parent window's IDCMP port (**Window.UserPort**). When using the IDCMP for input, the following IDCMP flags control how requester input events will be handled.

IDCMP_REQSET
With this flag set, the program will receive a message whenever a requester opens in its window. The application will receive one IDCMP_REQSET event for each requester opened in the window.

IDCMP_REQCLEAR
With this flag set, the program will receive a message whenever a requester is cleared from its window. The application will receive one IDCMP_REQCLEAR event for each requester closed in the window. By counting the number of IDCMP_REQSET and IDCMP_REQCLEAR events, the application may determine how many requesters are currently open in a window.

IDCMP_REQVERIFY

With this flag set, the application can ensure that it is ready to allow a requester to appear in the window before the requester is displayed.

When the program receives an IDCMP_REQVERIFY message, it must reply to that message before the requester is added to the window. If multiple requesters are opened in the window at the same time, only the first one will cause an IDCMP_REQVERIFY event. It is assumed that once a requester is in a window others may be added without the program's consent. After the requester count drops to zero and there are no open requesters in the window, the next requester to open will cause another IDCMP_REQVERIFY event.

IDCMP_REQVERIFY is ignored by the **Request()** function. Since **Request()** is controlled by the application, it is assumed that the program is prepared to handle the request when calling this function. Since the system does not render true requesters into an application's window (**EasyRequest()** and **AutoRequest()** come up in their own window, not in the application's window), IDCMP_REQVERIFY will only control the timing of double menu requesters.

These flags are set when the parent window is first opened by using either the WA_IDCMP tag or **NewWindow.IDCMPFlags**. They can also be set after the parent window is open by using the **ModifyIDCMP()** call. See the chapter entitled "Intuition Input and Output Methods," for further information about these IDCMP flags. See the "Intuition Windows" chapter for details on setting IDCMP flags when a window is opened.

Requester Structure

Unused fields in the **Requester** structure should be initialized to NULL or zero before using the structure. For global data that is pre-initialized, be sure to set all unused fields to zero. For dynamically allocated structures, allocate the storage with the MEMF_CLEAR flag, or call the **InitRequester()** function to clear the structure.

> *Requesters are Initialized According to Their Type.* See "Rendering Requesters" and "Gadgets in Requesters" above for information about how the initialization of the structure differs according to how the requester is rendered.

The specification for a **Requester** structure, defined in *<intuition/intuition.h>*, is as follows.

```
struct Requester
    {
    struct Requester *OlderRequest;
    WORD LeftEdge, TopEdge;
    WORD Width, Height;
    WORD RelLeft, RelTop;
    struct Gadget *ReqGadget;
    struct Border *ReqBorder;
    struct IntuiText *ReqText;
    UWORD Flags;
    UBYTE BackFill;
    struct Layer *ReqLayer;
    UBYTE ReqPad1[32];
    struct BitMap *ImageBMap;
    struct Window *RWindow;
    struct Image  *ReqImage;
    UBYTE ReqPad2[32];
    };
```

Here are the meanings of the fields in the **Requester** structure:

OlderRequest
For system use, initialize to **NULL**.

LeftEdge, TopEdge
The location of the requester relative to the upper left corner of the window. These values must be set if the POINTREL flag is not set. Use **RelLeft** and **RelTop** for POINTREL requesters.

Width, Height
These fields describe the size of the entire requester rectangle, containing all the text and gadgets.

RelLeft, RelTop
These values are only used if the POINTREL flag in the requester's **Flags** field is set.

If the requester is a double menu requester and POINTREL is set then these values contain the relative offset of the requester's upper left corner from the current pointer position.

If the requester is not a double menu requester and POINTREL is set, then these values contain the relative offset of the requester's center from the center of the window that the requester is to be displayed in. For example, using POINTREL with a requester which is not a double menu requester with **RelLeft** and **RelTop** of zero will center the requester in the window. The requester is centered within the inner part of the window, that is, within the inside edge of the window's borders.

If the requester is POINTREL and part of the containing box will appear out of the window, Intuition will adjust the requester so that the upper left corner is visible and as much of the remaining box as possible is visible. The adjustment attempts to maintain the requester within the window's borders, not within the window's bounding box.

ReqGadget
This field is a pointer to the first in a linked list of **Gadget** structures. GTYP_REQGADGET must be specified in the **GadgetTypes** field of all **Gadget** structures that are used in a requester. Take care not to specify gadgets that extend beyond the **Requester** rectangle specified by the **Width** and **Height** fields, as Intuition does no boundary checking.

For requesters with custom imagery, where PREDRAWN is set, **ReqGadget** points to a valid list of gadgets, which are real gadgets in every way except that the gadget text and imagery information are ignored (and can be NULL). The select box, highlighting, and gadget type data are still used. Try to maintain a close correspondence between the gadgets' select boxes and the supplied imagery.

String Gadgets and Pre-drawn Requesters. Intuition will not render string gadget text in a predrawn requester. The application must use other rendering means than the predrawn bitmap if it wishes to use string gadgets with a requester.

ReqBorder
This field is a pointer to an optional linked list of **Border** structures for drawing lines around and within the requester. The lines specified in the border may go anywhere in the requester; they are not confined to the perimeter of the requester.

For requesters with custom imagery, where PREDRAWN is set, this variable is ignored and may be set to NULL.

ReqText

This field is a pointer to an optional linked list of **IntuiText** structures containing text for the requester. This is for general text in the requester.

For requesters with custom imagery, where PREDRAWN is set, this variable is ignored and can be set to NULL.

Flags

The following flags may be specified for the **Requester**:

POINTREL

Specify POINTREL to indicate that the requester is to appear in a relative rather than a fixed position.

For double menu requesters, the position is relative to the pointer. Otherwise, the position of POINTREL requesters is relative to the center of the window.

See the discussion of **RelLeft** and **RelTop**, above.

PREDRAWN

Specify PREDRAWN if a custom **BitMap** structure is supplied for the requester and **ImageBMap** points to the structure.

NOISYREQ

Normally, when a requester is active, any gadget, menu, mouse and keyboard events within the parent window are blocked. Specify the NOISYREQ requester flag to allow keyboard and mouse button IDCMP events to be posted, even though the requester is active in the parent window.

If the NOISYREQ requester flag is set, the application will receive IDCMP_RAWKEY, IDCMP_VANILLAKEY and IDCMP_MOUSEBUTTONS events. Note that with NOISYREQ set, IDCMP_MOUSEBUTTON events will also be sent when the user clicks on any of the blocked gadgets in the parent window.

Although the reporting of mouse button events depends on NOISYREQ, mouse movement events do not. IDCMP_MOUSEMOVE events are reported if the window flag WFLG_REPORTMOUSE is set in the parent window, or if one of the requester gadgets is down and has the GACT_FOLLOWMOUSE flag set. This is true even if the requester is a double-menu requester.

USEREQIMAGE

Render the linked list of images pointed to by **ReqImage** after rendering the **BackFill** color but before gadgets and text.

NOREQBACKFILL

Do not backfill the requester with the **BackFill** pen.

In addition, Intuition uses these flags in the **Requester**:

REQOFFWINDOW
Set by Intuition if the requester is currently active but is positioned off window.

REQACTIVE
This flag is set or cleared by Intuition as the requester is posted and removed. The active requester is indicated by the value of **Window.FirstRequest**.

SYSREQUEST
This flag is set by Intuition if this is a system generated requester. Since the system will never create a true requester in an application window, the application should not be concerned with this flag.

BackFill
BackFill is the pen number to be used to fill the rectangle of the requester before any drawing takes place. For requesters with custom imagery, where PREDRAWN is set, or for requesters with NOREQBACKFILL set, this variable is ignored.

ReqLayer
While the requester is active, this contains the address of the **Layer** structure used in rendering the requester.

ImageBMap
A pointer to the custom bitmap for this requester. If this requester is not PREDRAWN, Intuition ignores this variable.

When a custom bitmap is supplied, the PREDRAWN flag in the requester's **Flags** field must be set.

RWindow
Reserved for system use.

ReqImage
A pointer to a list of **Image** structures used to create imagery within the requester. Intuition ignores this field if the flag USEREQIMAGE is not set. This imagery is automatically redrawn by Intuition each time the requester needs refreshing. The images are drawn after filling with the **BackFill** pen, but before the gadgets and text.

ReqPad1, ReqPad2
Reserved for system use.

Easy Requesters

EasyRequest() provides a simple way to make a requester that allows the user to select one of a limited number of choices. (A similar function, **AutoRequest()**, is also available but is not as flexible or as powerful. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for more information.)

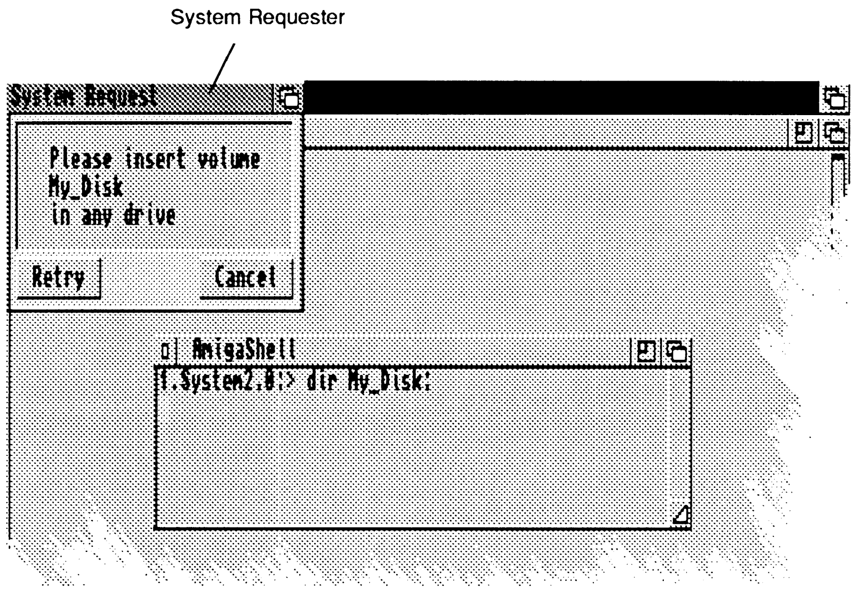


Figure 7-2: A Simple Requester Made with EasyRequest()

The program supplies the text for the body of the requester, text for each of the possible options, an optional title for the window, and other arguments. The body text can consist of one or more lines with lines separated by the linefeed character.

Each option for an easy requester is displayed as a simple button gadget positioned beneath the body text you specify. The layout of the requester, its text and buttons, is done automatically and is font sensitive. The screen font (**Screen.Font**) is used for all text in the requester.

Typically, easy requesters have one selection indicating a positive action and one selection indicating a negative action. The text used for the positive action might be "OK", "Yes," "True," "Retry," or similar responses. Likewise, the text used for the negative action might be "No," "False," "Cancel," and so on. The negative choice should always be the rightmost or final choice and will return a zero if selected.

When **EasyRequest()** is called, Intuition will build the requester, display it, and wait for user response.

```
LONG EasyRequest( struct Window *window, struct EasyStruct *easyStruct,
                  ULONG *idcmpPtr, APTR arg1, ... );

LONG EasyRequestArgs( struct Window *window, struct EasyStruct *easyStruct,
                      ULONG *idcmpPtr, APTR args );
```

The **window** argument is a pointer to the reference window. The requester will be displayed on the same screen that the reference window is on and also takes its title from the reference window, if not otherwise specified. This argument can be NULL, which means the requester is to appear on the Workbench screen,

The **easyStruct** argument is a pointer to an **EasyStruct** structure which defines the setup and the text of this easy requester (described below).

The **idcmpPtr** argument is a pointer to a ULONG containing the IDCMP flags for the event that you want to terminate this requester. If such an event occurs the requester is terminated (with a result of -1) and the ULONG that **idcmpPtr** points to will contain the actual class of the event message. This feature allows external events to satisfy the request, such as the user inserting a disk in the disk drive. This argument can be set to NULL for no automatic termination.

The gadget and body text for an easy requester is specified in an **EasyStruct** structure (see below). Body text can be specified using a **printf()**-style format string that also accepts variables as part of the text. If variables are specified in the requester text, their value is taken from the **args** (or **arg1,...**) parameters shown in the prototypes above. **EasyRequestArgs()** takes a pointer to an array of pointers to arguments, while **EasyRequest()** has a *varargs* interface and takes individual arguments as part of the function call. The types of these arguments are specified in the format strings of the **EasyStruct** structure. Arguments for **es_GadgetFormat** follow arguments for **es_TextFormat**.

The **EasyRequest()** functions return an integer from 0 to *n* - 1, where *n* is the number of choices specified for the requester. The numbering from left-to-right is: 1, 2, ..., *n* - 1, 0. This is for compatibility with **AutoRequest()** which returns FALSE for the rightmost gadget.

The function will return -1 if it receives an IDCMP event that matches one of the termination events specified in the **idcmpPtr** argument.

> *Turn Off the Verify Messages.* Use **ModifyIDCMP()** to turn off all verify messages (such as MENUVERIFY) before calling **EasyRequest()** or **AutoRequest()**. Neglecting to do so can cause situations where Intuition is waiting for the return of a message that the application program cannot receive because its input is shut off while the requester is up. If Intuition finds itself in a deadlock state, the verify function will timeout and will be automatically replied.

THE EASYSTRUCT STRUCTURE

The text and setup of an easy requester is specified in an **EasyStruct** structure, defined in *<intuition/intuition.h>*.

```
struct EasyStruct
    {
    ULONG       es_StructSize;
    ULONG       es_Flags;
    UBYTE       *es_Title;
    UBYTE       *es_TextFormat;
    UBYTE       *es_GadgetFormat;
    };
```

es_StructSize
: Contains the size of the **EasyStruct** structure, **sizeof(struct EasyStruct)**.

es_Flags
: Set to zero.

es_Title
: Title of easy requester window. If this is NULL, the title will be taken to be the same as the title of the reference window, if one is specified in the **EasyRequest()** call, else the title will be "System Request".

es_TextFormat

Format string for the text in the requester body, with **printf()**-style variable substitution as described in the Exec library function **RawDoFmt()**. Multiple lines are separated by the linefeed character (hex 0x0a or '\n' in C). Formatting '%' functions are supported exactly as in **RawDoFmt()**. The variables that get substituted in the format string come from the last argument passed to **EasyRequest()** (see prototype above).

es_GadgetFormat

Format string for gadgets, where the text for separate gadgets is separated by '|' (vertical bar). As with the body text, **printf()**-style formatting with variable substitution is supported, but multi-line text in the gadgets is not supported. At least one gadget *must* be specified.

Requesters generated with **EasyRequest()** and **BuildEasyRequest()** (including system requesters, which use **SysReqHandler()** to handle input) can be satisfied by the user via the keyboard. The key strokes left Amiga V and left Amiga B correspond to selecting the requester's leftmost or rightmost gadgets with the mouse, respectively.

An easy request must have at least one choice. Multiple choices are specified through the "|" (vertical bar) separator character in the **es_GadgetFormat** string. The buttons are displayed evenly spaced, from left-to-right in the order in which they appear in the string.

The requesters generated by **EasyRequest()** appear in the visible portion of the specified screen. They do not cause the screen to scroll. Under the current implementation, the window for an easy requester will appear in the upper left corner of the display clip for the specified screen.

When a request is posted using **EasyRequest()** or **BuildEasyRequest()**, it will move the screen it appears on to the front, if that screen is not already the frontmost. This brings the request to the attention of the user. The request also comes up as the active window and could potentially steal the input focus from the current window.

When the request is satisfied the screen will be moved to back if the request caused the screen to move to the front when it was displayed. Note that the final screen position may not be the same as the original screen position.

Example Using EasyRequest()

```
;/* easyrequest.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 easyrequest.c
Blink FROM LIB:c.o,easyrequest.o TO easyrequest LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** easyrequest.c - show the use of an easy requester.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
/* declare the easy request structure.
** this uses many features of EasyRequest(), including:
**     multiple lines of body text separated by '\n'.
**     variable substitution of a string (%s) in the body text.
**     multiple button gadgets separated by '|'.
**     variable substitution in a gadget (long decimal '%ld').
*/
struct EasyStruct myES =
    {
    sizeof(struct EasyStruct),
    0,
    "Request Window Name",
    "Text for the request\nSecond line of %s text\nThird line of text for the request",
    "Yes|%ld|No",
    };

struct Library *IntuitionBase;

/*
** Main routine to show the use of EasyRequest()
*/
VOID main (int argc, char **argv)
{
LONG answer;
LONG number;

number = 3125794;  /* for use in the middle button */

if (IntuitionBase = OpenLibrary("intuition.library",37))
    {
    /* note in the variable substitution:
    **     the string goes in the first open variable (in body text).
    **     the number goes in the second open (gadget text).
    */
    answer = EasyRequest(NULL, &myES, NULL, "(Variable)", number);

    /* Process the answer.  Note that the buttons are numbered in
    ** a strange order.  This is because the rightmost button is
    ** always a negative reply.  The code can use this if it chooses,
    ** with a construct like:
    **
    **     if (EasyRequest())
    **          positive_response();
    */
    switch (answer)
        {
        case 1:
            printf("selected 'Yes'\n");
            break;
        case 2:
            printf("selected '%ld'\n", number);
            break;
        case 0:
            printf("selected 'No'\n");
            break;
        }

    CloseLibrary(IntuitionBase);
    }
}
```

LOW LEVEL ACCESS TO EASY REQUESTERS

The **EasyRequest()** function calls a lower level Intuition function named **BuildEasyRequest()** to construct the requester. An application can call **BuildEasyRequest()** directly if it needs to use an easy requester but requires custom handling of the events sent to the requester. Handling of the events should be done using the **SysReqHandler()** function as described below.

The **BuildEasyRequest()** functions take the same arguments as **EasyRequest()**:

```
struct Window *BuildEasyRequestArgs( struct Window *window,
   struct EasyStruct *easyStruct, unsigned long idcmp, APTR args );

struct Window *BuildEasyRequest( struct Window *window,
    struct EasyStruct *easyStruct, unsigned long idcmp, APTR arg1, ... );
```

To process input event information directly while an easy requester is displayed, first call **BuildEasyRequest()** then call **SysReqHandler()** periodically to process user input.

```
LONG SysReqHandler( struct Window *window, ULONG *idcmpPtr, long waitInput );
```

This will provide standard handling of events but allow the application to control the timing of checking the events. This handling includes checks for left Amiga keys.

The **FreeSysRequest()** function must be called after an application has finished with a requester (if it was created with **BuildEasyRequest()** call).

```
void FreeSysRequest( struct Window *window );
```

This function ends the requester and frees any resources allocated with the **BuildEasyRequest()** call.

System Requesters

System requesters, such as DOS requests to "Insert volume foo in any drive," are created by the system using **EasyRequest()**. Unless otherwise specified, these requests appear on the default public screen.

System requests may appear at any time the system requires a resource that is not available. The user may be in the middle of an action, the program may be in any state.

Use the function **ModifyIDCMP()** to turn off all verify messages before calling any function that might generate a system requester. Neglecting to do so can cause situations where Intuition is waiting for the return of a message which the application program is unable to receive because its input is shut off while the requester is up. If Intuition finds itself in a deadlock state, the verify function will timeout and be automatically replied.

REDIRECTING SYSTEM REQUESTERS

A process can force the system requests which are caused by its actions to appear on a custom screen by changing the **pr_WindowPtr** field of its **Process** structure. This field may be set to three values: zero, negative one or a valid pointer to the **Window** structure of an open window. If **pr_WindowPtr** is set to zero, the request will appear on the default public screen. If **pr_WindowPtr** is set to negative one, the system request will never appear and the return code will be as if the user had selected the rightmost button (negative response). If **pr_WindowPtr** is set to a valid window pointer, then the request will appear on the same screen as the window.

The original value of **pr_WindowPtr** should be cached and restored before the window is closed.

Alerts

Alerts are for emergency messages. They can be displayed even when the system is in a very fragile state, such as when the system is low on memory or when some of the system lists are corrupt.

Alerts can be displayed by either the system or an application. They are reserved for urgent messages and dire warnings in situations that require the user to take some immediate action. Alerts should only be used where no other display type is possible. For instance, when the system has crashed or is about to crash, an alert could be used to inform the user of the cause.

The sudden display of an alert is a jarring experience for the user. The system stops dead while the alert is displayed and waits for the user input. For this reason, alerts should only be used when there is no recourse. If possible, use requesters or windows to display warning messages in place of alerts.

System alerts are managed entirely by Intuition. The program does not have to take any action to invoke or process these alerts. Alerts do not have access to the display database or other information required to open in specialized display modes. For this reason, alerts must appear in a display mode available on all machines, namely high resolution, non-interlaced. Alerts do not use overscan, so the display is limited to 640 by 200 on an NTSC machine, and 640 by 256 on a PAL machine.

The alert appears at the top of the video display. They are displayed the full 640 pixels wide and as tall as needed, up to the limits described above. Alerts are always displayed on a black background. The text of the alert is displayed within a rectangular border. Both the text and the border are displayed in a single color which is determined by the type of the alert.

The user responds to an alert by pressing one of the mouse buttons. The left mouse button signifies a positive response such as "Retry" or "OK". The right mouse button signifies a negative response such as "Cancel" or "Abort".

> *Alerts Save Up User Input.* The events produced by the user during an alert are not consumed by the alert. These events are passed through to the program when the alert returns. There could be a great deal of input queued and waiting for processing by the application.

TYPES OF ALERTS

There are two levels of severity for alerts:

RECOVERY_ALERT

Recovery alerts are used in situations where the caller believes that the system can resume operations after handling the error. The alert is used as a warning, and is displayed in amber.

A recoverable alert displays the text of the alert and flashes the border while waiting for the user to respond. It returns TRUE if the user presses the left mouse button in response to the alert, otherwise FALSE is returned.

DEADEND_ALERT

Deadend alerts are used in situations where the caller believes that no recovery from the error is possible, and further operation of the system is impossible. This alert is used to inform the user of a fatal problem and is displayed in red. Deadend alerts are the same as recoverable alerts in every way except color.

CREATING ALERTS

The function **DisplayAlert()** creates and displays an alert message. The message will almost always be displayed regardless of the state of the machine (with the exception of catastrophic hardware failures). If the user presses one of the mouse buttons, the display is restored to its original state, if possible. If a recoverable alert cannot be displayed (because memory is low), **DisplayAlert()** will return FALSE, as if the user had selected cancel. **DisplayAlert()** is also used by the system to display the Amiga system alert messages.

```
BOOL DisplayAlert( unsigned long alertNumber, UBYTE *string, unsigned long height );
```

The **alertNumber** argument is a LONG value, specifying whether this is a RECOVERY_ALERT or a DEADEND_ALERT (see the *<intuition/intuition.h>* include file).

The **string** argument points to a string that is made up of one or more substrings. Each substring contains the following:

- The first component is a 16 bit x-coordinate and an 8 bit y-coordinate describing where to position the substring within the alert display. The units are in pixels. The y-coordinate describes the location of the text baseline.
- The second component is the text itself. The substring must be NULL terminated (it ends with a zero byte).
- The last component is the continuation byte. If this byte is zero, this is the last substring in the message. If this byte is non-zero, there is another substring in this alert message.

The complete string must be terminated by two NULL characters; one as the end of the last substring, and one as a NULL continuation byte, indicating that this was the last substring. The **height** argument is the number of display lines required for the alert.

DISPLAY ALERT EXAMPLE

This program demonstrates an alert. An explanation of the positioning values for the alert strings is in the comment that precedes the **alertMsg** string.

```
;/* displayalert.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 displayalert.c
Blink FROM LIB:c.o,displayalert.o TO displayalert LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** displayalert.c -  This program implements a recoverable alert
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
/* Each string requires its own positioning information, as explained
** in the manual.  Hex notation has been used to specify the positions of
** the text. Hex numbers start with a backslash, an "x" and the characters
** that make up the number.
**
** Each line needs 2 bytes of X position, and 1 byte of Y position.
**   In our 1st line: x = \x00\xF0 (2 bytes) and y = \x14 (1 byte)
**   In our 2nd line: x = \x00\xA0 (2 bytes) and y = \x24 (1 byte)
** Each line is null terminated plus a continuation character (0=done).
** This example assumes that the compiler will concatenate adjacent
** strings into a single string with no extra NULLs.  The compiler does
** add the terminating NULL at the end of the entire string...The entire
** alert must end in TWO NULLs, one for the end of the string, and one
** for the NULL continuation character.
*/

UBYTE *alertMsg =
    "\x00\xF0\x14" "OH NO, NOT AGAIN!" "\x00\x01"
    "\x00\x80\x24" "PRESS MOUSEBUTTON:   LEFT=TRUE   RIGHT=FALSE" "\x00";

struct Library *IntuitionBase;

VOID main(int argc, char **argv)
{
if (IntuitionBase = OpenLibrary("intuition.library",33))
    {
    if (DisplayAlert(RECOVERY_ALERT, alertMsg, 52))
        printf("Alert returned TRUE\n");
    else
        printf("Alert returned FALSE\n");

    CloseLibrary(IntuitionBase);
    }
}
```

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of Intuition requesters and alerts. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 7-1: Functions for Intuition Requesters and Alerts

Function	Description
Request()	Open a requester in an open window.
EndRequest()	Close an open requester in a window.
InitRequester()	Clear a requester structure before use.
EasyRequestArgs()	Open a system requester.
EasyRequest()	Alternate calling sequence for **EasyRequestArgs()**.
BuildEasyRequestArgs()	Low level function to open **EasyRequestArgs()**.
BuildEasyRequest()	Low level function to close **EasyRequestArgs()**.
SysReqHandler()	Event handler function for **EasyRequestArgs()**.
AutoRequest()	Open a pre-V36 system requester.
BuildSysRequest()	Low level function to open an **AutoRequest()**.
FreeSysRequest()	Low level function to close an **AutoRequest()**.
SetDMRequest()	Set a double menu requester for an open window.
ClearDMRequest()	Clear a double menu requester from an open window.
DisplayAlert()	Open an alert on the screen.

Chapter 8
INTUITION IMAGES, LINE DRAWING AND TEXT

Intuition supports two general approaches to creating images, lines, and text in displays: through Intuition library calls and through graphics library calls.

This chapter explains the use of Intuition structures and functions for creating display imagery. The Intuition graphical functions provide a high level interface to the graphics library, giving the application quick and easy rendering capabilities. As with any high level calls, some power and flexibility is sacrificed in order to provide a simple interface.

For more flexibility and control over the graphics, the application can directly call functions in the graphics library as discussed in the ''Graphics Primitives'' chapter. Intuition also has additional features for defining custom graphic objects. See the ''BOOPSI'' chapter for more information on these objects.

Intuition Graphic Objects

Intuition graphic objects are easy to create and economical to use. There are just three basic types of graphic objects you can use yet these three types cover most rendering needs:

Image
: **Images** are graphic objects that can contain any imagery. They consist of a rectangular bitmap that can be any size and describes each individual pixel to be displayed.

Border
: **Borders** are connected lines of any length and number, drawn between an arbitrary series of points. They consist of a series of two dimensional coordinates that describe the points between which lines will be drawn.

IntuiText
: **IntuiText** strings are text strings of any length drawn in any font. They consist of a text string and font specification that describes the text to be rendered.

Each of these three objects may be chained together with other members of the same type. For instance, many lines of text may be rendered as a single object by linking many instances of **IntuiText** objects together. Only objects of the same type may be linked.

Any of these types can be rendered into any of the Intuition display elements (window, requester, menu, etc.). In fact, the application can often display the same structure in more than one position or more than one of the elements at the same time.

DISPLAYING IMAGES, BORDERS AND INTUITEXT

Images, **Borders** and **IntuiText** objects may be directly or indirectly rendered into the display by the application. The application can draw these objects directly into windows or screens by using one of the functions **DrawImage()**, **DrawBorder()** or **PrintIText()**. The application supplies the appropriate pointer to a **Border**, **Image** or **IntuiText** structure as an argument to the function, as well as position information and a pointer to the correct **RastPort**. These rendering functions are discussed in more detail below.

The application can also draw these objects indirectly by attaching them to a menu, gadget or requester. As Intuition places these elements on the display, it also renders the associated graphics. The **Requester**, **Gadget**, and **MenuItem** structures contain one or more fields reserved for rendering information. See the specific chapters on these items for information on attaching graphical objects to them.

POSITIONING GRAPHIC OBJECTS

The position of these objects is specified as the sum of two independent components: an external component which gives the position of a base reference point for the list of objects, and an internal component which gives the relative offset of a specific object to the base reference point.

The external component is used to position the object list within the display element. For objects drawn indirectly by attaching them to a menu, gadget or requester, this is always a point within the menu, gadget or requester (the top left corner).

For objects drawn directly with the **DrawImage()**, **DrawBorder()** or **PrintIText()** functions, specific x and y coordinates are provided as arguments that specify an offset within the screen's or window's **RastPort** at which to display the list of objects.

Each object also has an internal, relative component that is added to the external component described above to determine the final position of the object. This allows the application to reuse a graphical object and have it appear relative to each object to which it is attached. For example, if the application has numerous gadgets of the same size, it can use a single **Border** structure to draw lines around all the gadgets. When the gadgets are drawn, the base position of the lines will be taken from each specific gadget in turn.

Creating Images

With an **Image** structure an application can create graphic objects quickly and easily and display them almost anywhere. **Images** have an additional attribute that makes them even more economical—by changing two simple data elements in the **Image** structure, the color of the image may be changed.

Images are rectangular bitmaps which individually define the color of each pixel represented. Images may not be masked to allow part of the background to show through. The entire rectangular image is drawn into the target element, overwriting any data it may overlap. All bitplanes defined in the target **RastPort** within the image's rectangle are overwritten either with image data, ones or zeros.

Images may be directly drawn by the application by using the **DrawImage()** function, described below. The image may be rendered into any screen or window **RastPort** with this function. (**DrawImageState()** can also be used to draw the image, but this is an advanced topic discussed later in the ''BOOPSI'' chapter.

The visual imagery for an **Image** can be removed from the display by calling **EraseImage()**. For a normal **Image** structure, this will call the graphics function **EraseRect()**, which clears the **Image** rectangle by using the layer's backfill pen to overwrite it.

Alternately, images can be used indirectly by attaching them to menus, gadgets or requesters when they are initialized. For instance, in menus the **MenuItem** structure has the **ItemFill** and **SelectFill** fields. If the ITEMTEXT flag is cleared and the HIGHIMAGE flag is set, the application may place a pointer to a list of **Image** structures in each of these fields. The system will display the **ItemFill** images when the menu item is not selected and the **SelectFill** images when the menu item is selected. The application does not have to take any specific action to display these images. Once the menus have been added to a window, their management and display is under Intuition control.

The number of bitplanes in an image does not have to match the number of bitplanes in the display element in which the image is rendered. This provides great flexibility in using **Image** structures, as the same image may be reused in many places.

If the application's window is on the Workbench or some other public screen, it must use caution with hard-coded or constant image data, as the color palette of that screen is subject to change. If the application has its own custom screen, and it is appropriate for the colors of that screen to change, the same situation arises. Starting with V36, Intuition allows the screen opener to provide a mapping of pen number and rendering functions. For example, pens are specified for the bright and dark edges of three dimensional objects. Applications can obtain this mapping from the **DrawInfo** structure. See the ''Intuition Screens'' chapter for more information on **DrawInfo** and the new 3D look of Intuition in Release 2.

A suitably designed image may be drawn into a screen or window of any depth. To accomplish this, the application must ensure that detail is not lost when the image is displayed in a single bitplane **RastPort**, where only the first bitplane of image data will be displayed. This is important if the image will ever be displayed on the Workbench screen or any other public screen.

IMAGE STRUCTURE

For images, the application must create one or more instances of the **Image** structure.

```
struct Image
    {
    WORD LeftEdge;
    WORD TopEdge;
    WORD Width;
    WORD Height;
    WORD Depth;
    UWORD *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
    };
```

The meanings of the fields in the Image structure are:

LeftEdge, TopEdge

The location of the image relative to its base position when it is drawn. These offsets are added to the base position to determine the final location of the image data.

The base position for images rendered with **DrawImage()** is taken from arguments passed in the function call. For gadgets and menus, the base position is always the upper, left corner of the select box. For requesters the base position is always the upper, left corner of the requester.

Negative values of **LeftEdge** and **TopEdge** move the position up and to the left of the base position. Positive values move down and to the right.

Width, Height

The width and height of the image. **Width** contains the actual width of the image in pixels. **Height** specifies the height of the image in pixels.

The **Width** field of the **Image** structure contains the actual width in pixels of the widest part of the image, not how many pixels are contained in the words that define the image.

Depth

The depth of the image, or the number of bitplanes used to define it. This is not the depth of the screen or window in which the image will be displayed, it is the actual number of bitplanes that are defined in the **ImageData**.

ImageData

This is a pointer to the bits that define the image and determine the colors of each pixel. Image data must be placed in Chip memory. The data is organized as an array of 16 bit words whose size can be computed as follows:

```
WordWidth = ((Width + 16) / 16);
NumImageWords = WordWidth * Height * Depth;
```

The width of the image is rounded up to the nearest word (16 bits) and extra trailing bits are ignored. Each line of each bitplane must have enough words to contain the image width, with extra bits at the end of each line set to zero. For example, an image 7 bits wide requires one word for each line in the bitplane, whereas an image 17 bits wide requires two words for each line in the bitplane.

PlanePick

PlanePick tells which planes of the target **BitMap** are to receive planes of image data. This field is a bit-wise representation of bitplane numbers. For each bit set in **PlanePick**, there should be a corresponding bitplane in the image data.

PlaneOnOff

PlaneOnOff tells whether to set or clear bits in the planes in the target **BitMap** that receive no image data. This field is a bit-wise representation of bitplane numbers.

NextImage

This field is a pointer to another instance of an **Image** structure. Set this field to NULL if this is the last **Image** structure in the linked list.

DIRECTLY DRAWING THE IMAGE

As noted above, you use the **DrawImage()** call to directly draw an image into a screen or window **RastPort**.

```
void DrawImage( struct RastPort *rp, struct Image *image, long leftOffset, long topOffset );
```

The **rp** argument is a pointer to the **RastPort** into which the image should be drawn. This **RastPort** may come from a **Window** or **Screen** structure.

The **image** argument is a pointer to the list of **Image** structures that are to be rendered. The list may contain a single **Image** structure.

The **leftOffset** and **topOffset** arguments are the external component, or the base position, for this list of images. The **LeftEdge** and **TopEdge** values of each **Image** structure are added to these values to determine the final position of each image.

Images may also be indirectly drawn by attaching them to gadgets, menus or requesters when they are initialized.

IMAGE DATA

Image data must be in Chip memory. The **Image** structure itself may be in any memory, but the actual data referenced by **ImageData** field must be in Chip memory. This may be done by using compiler specific options, such as the __*chip* keyword of SAS/C, or by allocating memory with the MEMF_CHIP attribute and copying the image data to that memory.

Defining Image Data

Image data consists of binary data organized into a series of 16-bit words. The words must be sequential, where each successive word represents bits that are displayed later in the image. The image is defined as follows:

- The image is broken down into bitplanes. Each bitplane is considered separately.

- Within a single bitplane, each row of pixels is taken separately. First, round the number of pixels up to the next even multiple of 16. This determines the number of words used to represent a single row of data. For instance, an image that is 17 bits wide will require two 16-bit words to represent each row.

  The leftmost 16 pixel values are placed in the first word, followed by the next 16 pixel values, and so on. Any extra pixels at the end of the last word of the **ImageData** should be set to zero.

- The first row of data is the topmost row of the low order bitplane. This is immediately followed by the second row, then the third, until all rows in the bitplane have been represented.

- The data for the low order bitplane is followed immediately by the next to lowest, then the next, etc.

The color of each pixel in the image is directly related to the value in one or more memory bits, depending upon how many bitplanes there are in the image data and in which bitplanes of the screen or window the display is displayed.

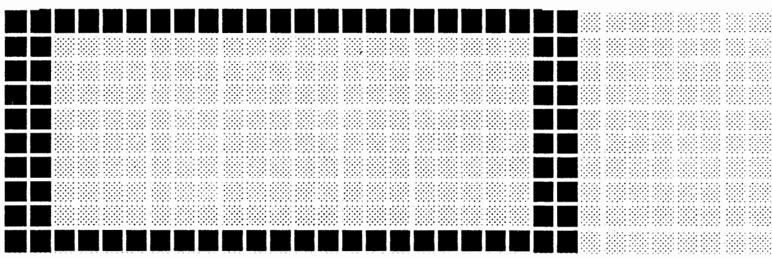
The color for a single pixel may be determined by combining the bits taken from the same relative position within each of the bitplanes used to define the image. For each pixel, the system combines all the bits in the same position to create a binary value that corresponds to one of the system color registers. This method of determining pixel color is called color indirection, because the actual color value is not in the display memory. Instead, it is in color registers that are located somewhere else in memory.

In many situations, the image and display will have different number of bitplanes, which complicates the determination of the color value for a given pixel. For now, assume that the image and display have the same number of bitplanes. The more complex example will be covered below, in the section ''Picking Bitplanes for the Image Display''.

If an image consists of only one bitplane and is displayed in a one bitplane display, then wherever there is a 0 bit in the image data, the color in color register zero is displayed and wherever there is a 1 bit, the color in color register one is displayed.

In an image composed of two bitplanes, the color of each pixel is obtained from a binary number formed by the values in two bits, one from the first bitplane and one from the second bitplane. If the bit in the first bitplane is a 1 and the bit in the second bitplane is a 0, then the color of that pixel will be taken from color register two (since 10 in binary is two in decimal). Again, the first bitplane describes all of the low order bits for each pixel. The second bitplane describes the next higher bit, and so on. This can be extended to any number of bitplanes.

Image Data



Hexadecimal Representation

F	F	F	F	F	F	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
C	0	0	0	0	3	0	0
F	F	F	F	F	F	0	0

Figure 8-1: Rendering of the Following Example Image

```
/* simpleimage.c - program to show the use of a simple Intuition Image.
**
** compiled with:
** lc -b1 -cfist -v -y simpleimage.c
** blink FROM LIB:c.o+"simpleimage.o" TO "simpleimage" LIB LIB:lc.lib LIB:amiga.lib
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif
```

```
struct IntuitionBase *IntuitionBase = NULL;

#define MYIMAGE_LEFT    (0)
#define MYIMAGE_TOP     (0)
#define MYIMAGE_WIDTH   (24)
#define MYIMAGE_HEIGHT  (10)
#define MYIMAGE_DEPTH   (1)

/* This is the image data.  It is a one bit-plane open rectangle which is 24
** pixels wide and 10 high.  Make sure that it is in CHIP memory, or allocate
** a block of chip memory with a call like this: AllocMem(data_size,MEMF_CHIP),
** and then copy the data to that block.  See the Exec chapter on Memory
** Allocation for more information on AllocMem().
*/
UWORD __chip myImageData[] =
    {
    0xFFFF, 0xFF00,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xFFFF, 0xFF00,
    };

/*
** main routine. Open required library and window and draw the images.
** This routine opens a very simple window with no IDCMP.  See the
** chapters on "Windows" and "Input and Output Methods" for more info.
** Free all resources when done.
*/
VOID main(int argc, char *argv[])
{
struct Window *win;
struct Image myImage;

IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library",37);
if (IntuitionBase != NULL)
    {
    if (NULL != (win = OpenWindowTags(NULL,
                        WA_Width,       200,
                        WA_Height,      100,
                        WA_RMBTrap,     TRUE,
                        TAG_END)))
        {
        myImage.LeftEdge    = MYIMAGE_LEFT;
        myImage.TopEdge     = MYIMAGE_TOP;
        myImage.Width       = MYIMAGE_WIDTH;
        myImage.Height      = MYIMAGE_HEIGHT;
        myImage.Depth       = MYIMAGE_DEPTH;
        myImage.ImageData   = myImageData;
        myImage.PlanePick   = 0x1;              /* use first bit-plane     */
        myImage.PlaneOnOff  = 0x0;              /* clear all unused planes */
        myImage.NextImage   = NULL;

        /* Draw the 1 bit-plane image into the first bit-plane (color 1) */
        DrawImage(win->RPort,&myImage,10,10);

        /* Draw the same image at a new location */
        DrawImage(win->RPort,&myImage,100,10);

        /* Wait a bit, then quit.
        ** In a real application, this would be an event loop, like the
        ** one described in the Intuition Input and Output Methods chapter.
        */
        Delay(200);

        CloseWindow(win);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}
```

PICKING BITPLANES FOR IMAGE DISPLAY

A single image may be displayed in different colors without changing the underlying image data. This is done by selecting which of the target bitplanes are to receive the image data, and what to do with the target bitplanes that do not receive any image data.

PlanePick and **PlaneOnOff** are used to control the bitplane rendering of the image. The bits in each of these variables have a direct correspondence to the bitplanes of the target bitmap. The lowest bit position corresponds to the lowest numbered bitplane, the next highest bit position corresponds to the next bitplane, etc.

For example, for a window or screen with three bitplanes (consisting of planes 0, 1, and 2), all the possible values for **PlanePick** or **PlaneOnOff** and the planes picked are as follows:

PlanePick or PlaneOnOff	Planes Picked
000	No planes
001	Plane 0
010	Plane 1
011	Planes 0 and 1
100	Plane 2
101	Planes 0 and 2
110	Planes 1 and 2
111	Planes 0, 1, and 2

PlanePick picks the bitplanes of the containing **RastPort** that will receive the bitplanes of the image. For each plane that is picked to receive data, the next successive plane of image data is drawn there. For example, if an image with two bitplanes is drawn into a window with four bitplanes with a **PlanePick** of binary 1010, the first bitplane of the image will be drawn into the second bitplane of the window and the second bitplane of the image will be drawn into the fourth bitplane of the window. Do not set more bits in **PlanePick** than there are bitplanes in the image data.

PlaneOnOff specifies what to do with the bitplanes that are not picked to receive image data. If the **PlaneOnOff** bit is zero, then the associated bitplane will be filled with zeros. If the **PlaneOnOff** bit is one, then the associated bitplane will be filled with ones. Of course, only bits that fall within the rectangle defined by the image are affected by this manipulation.

Only the bits not set in **PlanePick** are used in **PlaneOnOff**, that is, **PlaneOnOff** only applies to those bitplanes not picked to receive image data. For example, if **PlanePick** is 1010 and **PlaneOnOff** is 1100, then **PlaneOnOff** may be viewed as x1x0 (where the *x* positions are not taken into consideration). In this case, planes two and four would receive image data and planes one and three would be set by **PlaneOnOff**. Each bit in plane one would be set to zero and each bit in plane three would be set to one.

PlaneOnOff is only useful where an entire bitplane of an image may be set to the same value. If the bitplane is not all set to the same value, even for just a few bits, then image data must be specified for that plane.

A simple trick to create a filled rectangle of any color may be used by supplying no image data, where the color is controlled by **PlaneOnOff**. The **Depth** of such an image is set to zero, the size of the rectangle is specified in the **Width** and **Height** fields and the **ImageData** pointer may be NULL. **PlanePick** should be set to zero, as there are no planes of image data to pick. **PlaneOnOff** is then set to the color register which contains the desired color for the rectangle.

IMAGE EXAMPLE

A more complex example of the use of an **Image** is presented below.

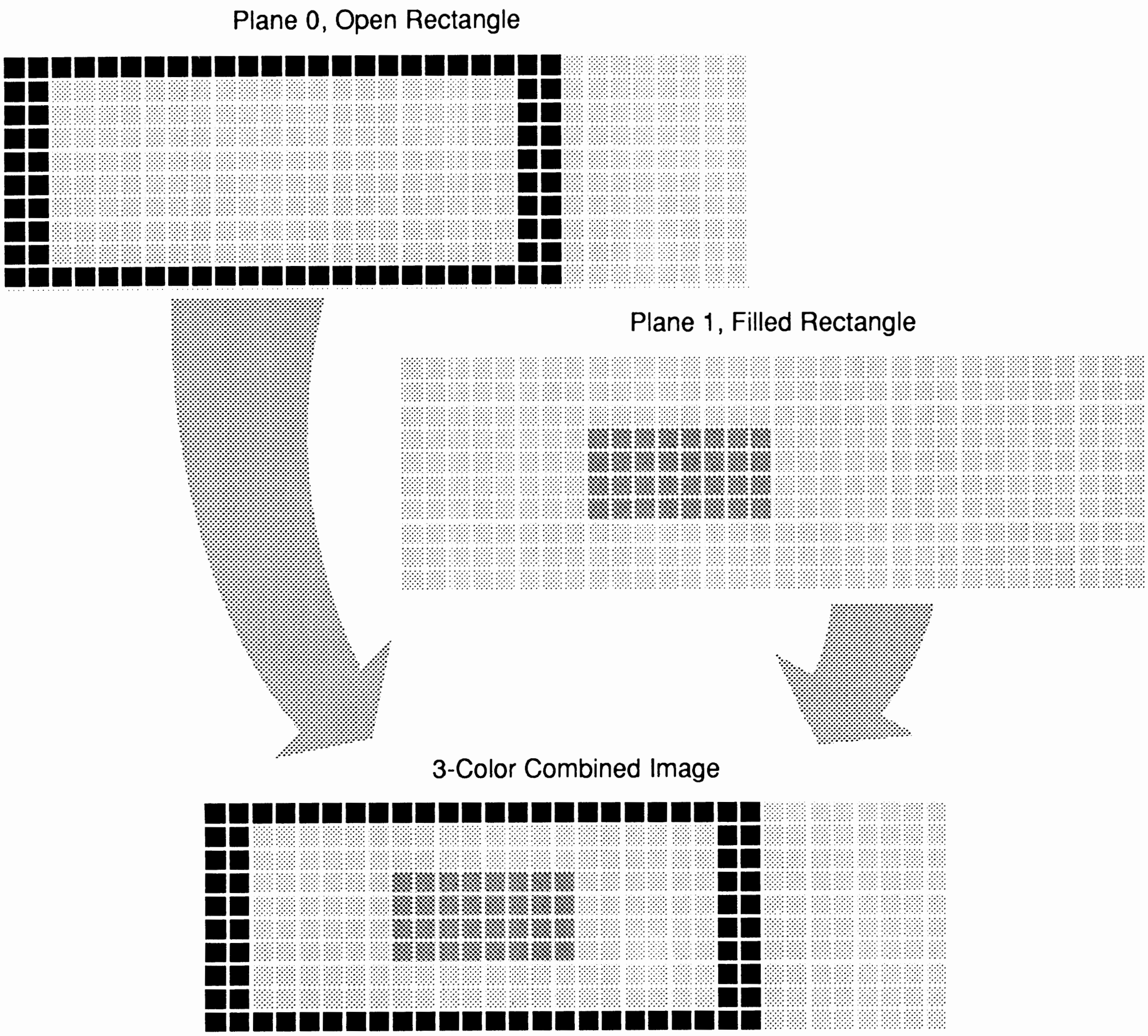


Figure 8-2: Picture of the More Complex Example Image

```
/* compleximage.c - program to show the use of a complex Intuition Image.
**
** compiled with:
** lc -b1 -cfist -v -y compleximage.c
** blink FROM LIB:c.o compleximage.o TO compleximage LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct IntuitionBase *IntuitionBase = NULL;

#define MYIMAGE_LEFT    (0)
#define MYIMAGE_TOP     (0)
#define MYIMAGE_WIDTH   (24)
#define MYIMAGE_HEIGHT  (10)
#define MYIMAGE_DEPTH   (2)

/* This is the image data.  It is a two bitplane open rectangle which
** is 24 pixels wide and 10 high.  Make sure that it is in CHIP memory,
** or allocate a block of chip memory with a call like:
**
**     AllocMem(data_size,MEMF_CHIP)
**
** and copy the data to that block.  See the Exec chapter on
** Memory Allocation for more information on AllocMem().
*/
UWORD __chip myImageData[] =
    {
    /* first bitplane of data,
    ** open rectangle.
    */
    0xFFFF, 0xFF00,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xC000, 0x0300,
    0xFFFF, 0xFF00,

    /* second bitplane of data,
    ** filled rectangle to appear within open rectangle.
    */
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x00FF, 0x0000,
    0x00FF, 0x0000,
    0x00FF, 0x0000,
    0x00FF, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    0x0000, 0x0000,
    };

/* used to get the "new look" on a custom screen */
UWORD pens[] = { ~0 };
```

```
/*
** main routine. Open required library and window and draw the images.
** This routine opens a very simple window with no IDCMP.  See the
** chapters on "Windows" and "Input and Output Methods" for more info.
** Free all resources when done.
*/
VOID main(int argc, char *argv[])
{
struct Screen *scr;
struct Window *win;
struct Image myImage;

IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library",37);
if (IntuitionBase != NULL)
    {
    if (NULL != (scr = OpenScreenTags(NULL,
                        SA_Depth,       4,
                        SA_Pens,        &pens,
                        TAG_END)))
        {
        if (NULL != (win = OpenWindowTags(NULL,
                            WA_RMBTrap,      TRUE,
                            WA_CustomScreen, scr,
                            TAG_END)))
            {
            myImage.LeftEdge    = MYIMAGE_LEFT;
            myImage.TopEdge     = MYIMAGE_TOP;
            myImage.Width       = MYIMAGE_WIDTH;
            myImage.Height      = MYIMAGE_HEIGHT;
            myImage.Depth       = MYIMAGE_DEPTH;
            myImage.ImageData   = myImageData;
            myImage.PlanePick   = 0x3;              /* use first two bitplanes */
            myImage.PlaneOnOff  = 0x0;              /* clear all unused planes  */
            myImage.NextImage   = NULL;

            /* Draw the image into the first two bitplanes */
            DrawImage(win->RPort,&myImage,10,10);

            /* Draw the same image at a new location */
            DrawImage(win->RPort,&myImage,100,10);

            /* Change the image to use the second and fourth bitplanes,
            ** PlanePick is 1010 binary or 0xA,
            ** and draw it again at a different location
            */
            myImage.PlanePick = 0xA;
            DrawImage(win->RPort,&myImage,10,50);

            /* Now set all the bits in the first bitplane with PlaneOnOff.
            ** This will make all the bits set in the second bitplane
            ** appear as color 3 (0011 binary), all the bits set in the
            ** fourth bitplane appear as color 9 (1001 binary) and all
            ** other pixels will be color 1 (0001 binary.  If there were
            ** any points in the image where both bits were set, they
            ** would appear as color 11 (1011 binary).
            ** Draw the image at a different location.
            */
            myImage.PlaneOnOff = 0x1;
            DrawImage(win->RPort,&myImage,100,50);

            /* Wait a bit, then quit.
            ** In a real application, this would be an event loop, like the
            ** one described in the Intuition Input and Output Methods chapter.
            */
            Delay(200);

            CloseWindow(win);
            }
        CloseScreen(scr);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}
```

Creating Borders

This data type is called a **Border** since it was originally used to create border lines around display objects. It is actually a general purpose structure for drawing connected lines between any series of points.

A **Border** is easier to use than an **Image** structure. Only the following need be specified to define a border:

- An internal position component which is used in determining the final position of the border.
- A set of coordinate pairs for each vertex.
- A color for the lines.
- One of several drawing modes.

BORDER STRUCTURE DEFINITION

To use a border, the application must create one or more instances of the **Border** structure. Here is the specification:

```
struct Border
    {
    WORD LeftEdge, TopEdge;
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    BYTE Count;
    WORD *XY;
    struct Border *NextBorder;
    };
```

Here is a brief description of the fields of the **Border** structure.

LeftEdge, TopEdge
These fields are used to determine the position of the **Border** relative to its base position (the base position is the upper left corner for requesters, menus, or gadgets and is specified in the call to **DrawBorder()** for windows and screens).

FrontPen, BackPen
These fields contain color registers numbers. **FrontPen** is the color used to draw the lines. **BackPen** is currently unused.

DrawMode
Set the **DrawMode** field to one of the following:

JAM1
Use **FrontPen** to draw the line.

COMPLEMENT
Change the pixels within the lines to their complement color.

Count
Specify the number of data points used in this border. Each data point is described by two words of data in the **XY** array.

XY A pointer to an array of coordinate pairs, one pair for each point. These coordinates are measured relative to the position of the border.

NextBorder
This field is a pointer to another instance of a **Border** structure. Set this field to NULL if this is the last **Border** structure in the linked list.

DIRECTLY DRAWING THE BORDERS

Borders may be directly drawn by the application by calling the function **DrawBorder()**.

```
void DrawBorder( struct RastPort *rp, struct Border *border, long leftOffset, long topOffset );
```

The **rp** argument is a pointer to the **RastPort** into which the border should be drawn. This rastport may come from a **Window** or **Screen** structure.

The **border** argument is a pointer to a list of **Border** structures which are to be rendered. The list may contain a single **Border** structure.

The **leftOffset** and **topOffset** arguments are the external component, or base position, for this list of **Borders**. The **LeftEdge** and **TopEdge** values of each **Border** structure are added to these to determine the **Border** position.

Borders may also be indirectly drawn by attaching them to gadgets, menus or requesters.

BORDER EXAMPLE

The following example draws a double border using two pens to create a shadow effect. The border is drawn in two positions to show the flexibility in positioning borders, note that it could also be attached to a menu, gadget or requester.

```
;/* shadowborder.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 shadowborder.c
Blink FROM LIB:c.o,shadowborder.o TO shadowborder LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** shadowborder.c - program to show the use of an Intuition Border.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library *IntuitionBase = NULL;

#define MYBORDER_LEFT    (0)
#define MYBORDER_TOP     (0)
```

```
/* This is the border data. */
WORD myBorderData[] =
    {
    0,0, 50,0, 50,30, 0,30, 0,0,
    };


/*
** main routine. Open required library and window and draw the images.
** This routine opens a very simple window with no IDCMP.  See the
** chapters on "Windows" and "Input and Output Methods" for more info.
** Free all resources when done.
*/
VOID main(int argc, char **argv)
{
struct Screen   *screen;
struct DrawInfo *drawinfo;
struct Window   *win;
struct Border    shineBorder;
struct Border    shadowBorder;

ULONG mySHADOWPEN = 1;  /* set default values for pens */
ULONG mySHINEPEN  = 2;  /* in case can't get info...   */

IntuitionBase = OpenLibrary("intuition.library",37);
if (IntuitionBase)
    {
    if (screen = LockPubScreen(NULL))
        {
        if (drawinfo = GetScreenDrawInfo(screen))
            {
            /* Get a copy of the correct pens for the screen.
            ** This is very important in case the user or the
            ** application has the pens set in a unusual way.
            */
            mySHADOWPEN = drawinfo->dri_Pens[SHADOWPEN];
            mySHINEPEN  = drawinfo->dri_Pens[SHINEPEN];

            FreeScreenDrawInfo(screen,drawinfo);
            }
        UnlockPubScreen(NULL,screen);
        }

    /* open a simple window on the workbench screen for displaying
    ** a border.  An application would probably never use such a
    ** window, but it is useful for demonstrating graphics...
    */
    if (win = OpenWindowTags(NULL,
                        WA_PubScreen,  screen,
                        WA_RMBTrap,      TRUE,
                        TAG_END))
        {
        /* set information specific to the shadow component of the border */
        shadowBorder.LeftEdge   = MYBORDER_LEFT + 1;
        shadowBorder.TopEdge    = MYBORDER_TOP + 1;
        shadowBorder.FrontPen   = mySHADOWPEN;
        shadowBorder.NextBorder = &shineBorder;

        /* set information specific to the shine component of the border */
        shineBorder.LeftEdge    = MYBORDER_LEFT;
        shineBorder.TopEdge     = MYBORDER_TOP;
        shineBorder.FrontPen    = mySHINEPEN;
        shineBorder.NextBorder  = NULL;

        /* the following attributes are the same for both borders. */
        shadowBorder.BackPen    = shineBorder.BackPen   = 0;
        shadowBorder.DrawMode   = shineBorder.DrawMode  = JAM1;
        shadowBorder.Count      = shineBorder.Count     = 5;
        shadowBorder.XY         = shineBorder.XY        = myBorderData;

        /* Draw the border at 10,10 */
        DrawBorder(win->RPort,&shadowBorder,10,10);

        /* Draw the border again at 100,10 */
        DrawBorder(win->RPort,&shadowBorder,100,10);
```

```
        /* Wait a bit, then quit.
        ** In a real application, this would be an event loop, like the
        ** one described in the Intuition Input and Output Methods chapter.
        */
        Delay(200);

        CloseWindow(win);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

BORDER COLORS AND DRAWING MODES

Borders can select their colors from the values set in the color registers for the screen in which they are rendered. The available number of colors and palette settings are screen attributes and may not be changed through border rendering.

Two drawing modes pertain to border lines: JAM1 and COMPLEMENT. To draw the line in a specific color, use the JAM1 draw mode. This mode converts each pixel in the line to the color set in the **FrontPen** field.

Selecting the COMPLEMENT draw mode causes the line to be drawn in an exclusive-or mode that inverts the color of each pixel within the line. The data bits of the pixel are changed to their binary complement. This complement is formed by reversing all bits in the binary representation of the color register number. In a three bitplane display, for example, color 6 is 110 in binary. In COMPLEMENT draw mode, if a pixel is color 6, it will be changed to the 001 (binary), which is color 1. Note that a border drawn in COMPLEMENT mode can be removed from a static display by drawing the border again in the same position.

BORDER COORDINATES

Intuition draws lines between points that are specified as sets of X, Y coordinates. Border data does not have to be in Chip memory.

The **XY** field contains a pointer to an array of coordinate pairs. All of these coordinates are offsets relative to the **Border** position, which is determined by the sum of the external and internal position components as described above. The coordinate pairs are ordered sequentially. The first two numbers make up the first coordinate pair, the next two numbers make up the second pair, and so on. Within a coordinate pair, the first number is the X offset and the second number is the Y offset.

The first coordinate pair describes the starting point of the first line. When the **Border** is rendered, a line is drawn between each pair of points. The first line is drawn from point one to point two, the second line is drawn from point two to point three, and so on, until the final point is reached.

The numbers specified in the **XY** array may be positive or negative. Negative values move up and to the left relative to the **Border** position, positive values move down and to the right. Again, the **Border** position is determined by adding the external position component and the internal position component. For example, a **Border** attached to a **Gadget** has an external component equal to the upper left corner of the gadget's select box. The internal component is set within the **Border** structure itself. These two components are added together and offsets from the resulting position, specified within the **XY** array, determine where the lines of the **Border** will appear.

Suppose the top left corner of the select box of the gadget is at window position (10,5). If the **Border** has **LeftEdge** set to 10 and **TopEdge** set to 10, then the **Border** is positioned at (10+10,5+10), that is (20,15). All **XY** coordinates will be relative to this **Border** position. If the **XY** array contains '0,5, 15,5, 15,0', then the relative coordinates will be (0,5), (15,5) and (15,0). Adding each coordinate to the **Border** position gives the absolute position of the lines within the window. This **Border** will draw two lines in the window, one from (20,20) to (35,20) and the second from (35,20) to (35,15).

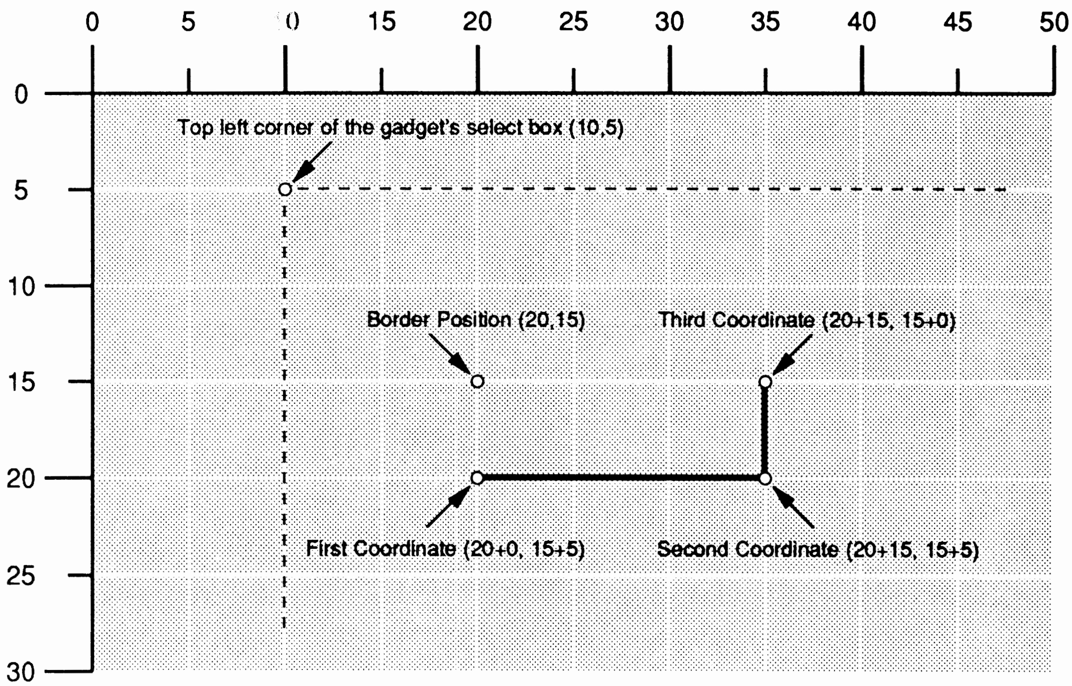


Figure 8-3: Example of Border Relative Position

To create a border that is outside the select box of a gadget, specify negative values in the internal component or use negative values for the initial **XY** values. For example, setting **LeftEdge** to -1 and **TopEdge** to -1 moves the position of the **Border** one pixel above and one pixel to the left of the gadget's select box.

LINKING BORDERS

The **NextBorder** field can point to another instance of a **Border** structure. This allows complex graphic objects to be created by linking together **Border** structures, each with its own data points, color and draw mode. This might be used, for instance, to draw a double border around a requester or gadget where the outer border is a second **Border** structure, linked to the first inner border.

Note that the borders can share data. For instance, to create a border with a shadow, link two borders together each of which points to the same **XY** data. Set the first border to draw in a dark pen (such as the SHADOWPEN from the screen's **DrawInfo** structure) and position the border down and to the right a few pixels by changing **LeftEdge** and **TopEdge** in the **Border** structure.

The second border should be set to a bright pen (such as the SHINEPEN in the screen's **DrawInfo** structure). When the border is drawn, the first border will draw in a dark color and then the second border will be drawn over it in a light color. Since they use the same data set, and the dark border is shifted down and to the right, the border will have a three dimensional appearance. This technique is demonstrated in the example listed earlier in this section.

Creating Text

The **IntuiText** structure provides a simple way of writing text strings within an Intuition display element. These strings may be used in windows, screens, menus, gadgets and requesters. To set up an **IntuiText**, you specify the following:

- Pen colors for the text.
- A draw mode.
- The starting offset for the text.
- The font used to render the text.
- The text string to output.

INTUITEXT STRUCTURE

To render text using Intuition, the application must create one or more instances of the **IntuiText** structure:

```
struct IntuiText
    {
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    WORD LeftEdge;
    WORD TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
    };
```

Here is a brief description of each member of the **IntuiText** structure:

FrontPen
The pen number specifying the color used to draw the text.

BackPen
The pen number specifying the color used to draw the background for the text, if JAM2 drawing mode is specified.

DrawMode
This field specifies one of four drawing modes:

JAM1
FrontPen is used to draw the text; background color is unchanged.

JAM2
FrontPen is used to draw the text; background color is changed to the color in **BackPen**.

COMPLEMENT
The characters are drawn in the complement of the colors that were in the background.

INVERSVID
Inverses the draw modes describe above. For instance INVERVID used with JAM1 means the character is untouched while the background is filled with the color of the **FrontPen**.

LeftEdge and **TopEdge**
The location of the text relative to its base position when it is drawn. These offsets are added to the base position to determine the final location of the text data.

The base position for text rendered with **PrintIText()** is taken from arguments passed in the function call. For gadgets and menus, the base position is always the upper, left corner of the select box. For requesters the base position is always the upper, left corner of the requester.

LeftEdge gives the offset of the left edge of the character cell and **TopEdge** gives the offset of the top edge of the character cell for the first character in the text string. Negative values of **LeftEdge** and **TopEdge** move the position up and to the left of the base position. Positive values move down and to the right.

ITextFont
A pointer to a **TextAttr** structure defining the font to be used. Set this to NULL to use the default font.

IText
A pointer to the NULL terminated text string to be displayed.

NextText
A pointer to another instance of **IntuiText**. Set this field to NULL for the last **IntuiText** in a list.

DIRECTLY DRAWING THE INTUITEXT

Use the **PrintIText()** call to directly draw the text into the target **RastPort** of a window or screen.

```
void PrintIText( struct RastPort *rp, struct IntuiText *iText, long left, long top );
```

The **rp** argument is a pointer to the **RastPort** into which the text should be drawn. This **RastPort** can come from a **Window** or **Screen** structure.

The **iText** argument is a pointer to a list of **IntuiText** structures which are to be rendered. The list may contain a single **IntuiText** structure. If the font is not specified in the **IntuiText** structure, Intuition will render the text using the **RastPort**'s font.

The **left** and **top** arguments give the external component, or base position for this list of **IntuiText** structures. The **LeftEdge** and **TopEdge** values in each **IntuiText** structure are added to these to determine the final position of the text.

IntuiText objects may also be drawn indirectly by attaching them to gadgets, menus or requesters.

DETERMINING TEXT LENGTH

To determine the pixel length of a given **IntuiText** string, call the **IntuiTextLength()** function.

```
LONG IntuiTextLength( struct IntuiText *iText );
```

Set the **iText** argument to point to the **IntuiText** structure whose length is to be found. This function will return the length of the **iText** text string in pixels. Note that if the **ITextFont** field of the given **IntuiText** is set to NULL, or Intuition cannot access the specified font, then **GfxBase->DefaultFont** will be used in determining the length of the text. This may not be the same as the **RastPort** font with which the text would be printed.

INTUITEXT EXAMPLE

```
;/* intuitext.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 intuitext.c
Blink FROM LIB:c.o,intuitext.o TO intuitext LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** intuitext.c - program to show the use of an Intuition IntuiText object.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

struct Library *IntuitionBase = NULL;

#define MYTEXT_LEFT (0)
#define MYTEXT_TOP  (0)

/*
** main routine. Open required library and window and draw the images.
** This routine opens a very simple window with no IDCMP.  See the
** chapters on "Windows" and "Input and Output Methods" for more info.
** Free all resources when done.
*/
VOID main(int argc, char **argv)
{
struct Screen    *screen;
struct DrawInfo  *drawinfo;
struct Window    *win;
struct IntuiText  myIText;
struct TextAttr   myTextAttr;

ULONG myTEXTPEN;
ULONG myBACKGROUNDPEN;

IntuitionBase = OpenLibrary("intuition.library",37);
if (IntuitionBase)
    {
    if (screen = LockPubScreen(NULL))
        {
        if (drawinfo = GetScreenDrawInfo(screen))
            {
            /* Get a copy of the correct pens for the screen.
            ** This is very important in case the user or the
            ** application has the pens set in a unusual way.
```

```
        */
        myTEXTPEN = drawinfo->dri_Pens[TEXTPEN];
        myBACKGROUNDPEN  = drawinfo->dri_Pens[BACKGROUNDPEN];

        /* create a TextAttr that matches the specified font. */
        myTextAttr.ta_Name  = drawinfo->dri_Font->tf_Message.mn_Node.ln_Name;
        myTextAttr.ta_YSize = drawinfo->dri_Font->tf_YSize;
        myTextAttr.ta_Style = drawinfo->dri_Font->tf_Style;
        myTextAttr.ta_Flags = drawinfo->dri_Font->tf_Flags;

        /* open a simple window on the workbench screen for displaying
        ** a text string.  An application would probably never use such a
        ** window, but it is useful for demonstrating graphics...
        */
        if (win = OpenWindowTags(NULL,
                            WA_PubScreen,    screen,
                            WA_RMBTrap,      TRUE,
                            TAG_END))
            {
            myIText.FrontPen    = myTEXTPEN;
            myIText.BackPen     = myBACKGROUNDPEN;
            myIText.DrawMode    = JAM2;
            myIText.LeftEdge    = MYTEXT_LEFT;
            myIText.TopEdge     = MYTEXT_TOP;
            myIText.ITextFont   = &myTextAttr;
            myIText.IText       = "Hello, World.  ;-)";
            myIText.NextText    = NULL;

            /* Draw the text string at 10,10 */
            PrintIText(win->RPort,&myIText,10,10);

            /* Wait a bit, then quit.
            ** In a real application, this would be an event loop,
            ** like the one described in the Intuition Input and
            ** Output Methods chapter.
            */
            Delay(200);

            CloseWindow(win);
            }
        FreeScreenDrawInfo(screen,drawinfo);
        }
    UnlockPubScreen(NULL,screen);
    }
CloseLibrary(IntuitionBase);
}
}
```

TEXT COLORS AND DRAWING MODES

IntuiText gets its colors from the values set in the color registers for the screen in which they are rendered. The available number of colors and palette settings are screen attributes and cannot be changed through IntuiText rendering.

Text characters in general are made up of two areas: the character image itself and the background area surrounding the character image. The color used in each area is determined by the draw mode which can be set to JAM1, JAM2 or COMPLEMENT. The flag INVERSVID may also be specified.

JAM1 draw mode renders each character with **FrontPen** and leaves the background area unaffected. Because the background of a character is not drawn, the pixels of the destination memory around the character image are not disturbed. Graphics beneath the text will be visible in the background area of each character cell.

JAM2 draw mode renders each character with **FrontPen** and renders each character background with **BackPen**. Using this mode, any graphics that previously appeared beneath the character cells will be totally overwritten.

COMPLEMENT draw mode renders the pixels of each character as the binary complement of the color that is currently at the destination pixel. The destination is the display memory where the text is drawn. As with JAM1, nothing is drawn into the background. **FrontPen** and **BackPen** are not used in COMPLEMENT mode. To determine the complement color, invert all the bits in the binary representation of the color register number. The resulting number specifies the color register to use for that pixel. In a three bitplane display, for example, color 6 (110 in binary) is the complement of color 1 (001 in binary).

The INVERSVID flag inverses the video for each of the drawing modes. For JAM1, nothing is drawn into the character area and the background is drawn in **FrontPen**. For JAM2, the character area is drawn in **BackPen** and the background is drawn in **FrontPen**. For COMPLEMENT mode, nothing is drawn into the character area and the background is complemented.

FONTS

The application may choose to specify the font used in rendering the **IntuiText**, or it may choose to use the default font for the system.

To use the default font, set the **ITextFont** field to NULL. Some care must be taken when using the default font. When an **IntuiText** object is rendered and no font is specified, the text will be rendered in the font set in the **RastPort**.

If the **RastPort** font is NULL, the text will be rendered using **GfxBase->DefaultFont**. Also, **IntuiTextLength()** always uses **GfxBase->DefaultFont** when **ITextFont** is NULL. The application must have open the graphics library in order to check the default font in **GfxBase**. (See the graphics library chapter for more information.)

To use a specific font for this text, place a pointer to an initialized **TextAttr** structure in the **ITextFont** field. Intuition will only use the specified font if it is available through a call to the **OpenFont()** routine. To use a font from disk, the application must first open the font using the **OpenDiskFont()** function. For more information about using fonts, see the "Graphics Library and Text" chapter in this manual.

LINKING TEXT STRINGS

The **NextText** field can point to another instance of an **IntuiText** structure. This allows the application to create a complex object which has several distinct groups of characters, each with its own color, font, location, and drawing mode. This can be used to create multiple lines of text, to position characters in the text very accurately and to change the color or font of the text. Each list of **IntuiText** objects may be drawn with one call to **PrintIText()**, or attached to a gadget, menu or requester as a single object.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of graphics under Intuition. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 8-1: Functions for Intuition Drawing Capabilities

Function	Description
DrawBorder()	Draw a border into a rast port.
DrawImage()	Draw a image into a rast port.
PrintIText()	Draw Intuition text into a rast port.
IntuiTextLength()	Find the length of an IntuiText string.
BeginRefresh()	Begin optimized rendering after a refresh event.
EndRefresh()	End optimized rendering after a refresh event.
GetScreenDrawInfo()	Get screen drawing information (V36).
FreeScreenDrawInfo()	Free screen drawing information (V36).

Chapter 9
INTUITION INPUT AND OUTPUT METHODS

This chapter discusses the input and output (I/O) techniques used with Intuition. I/O facilities are also available through Exec's device subsystems, such as the console, serial and parallel devices. (For more information on these see the *Amiga ROM Kernel Reference Manual: Devices*.)

For graphical output to the Amiga's display, programs can use Intuition's drawing features or handle rendering directly through calls to the graphics library. See the three graphics library chapters in this manual for more information on display rendering. For more about Intuition's drawing features see the chapter on "Intuition Images, Line Drawing and Text".

Overview of System I/O

This section provides a very simplified model of how Amiga I/O and application programs interact. The main elements of the Amiga's I/O system are shown in the diagram below. Input events begin when mouse movement is detected by the gameport device or key presses are received by the keyboard device. These and other input events are merged into a single stream by the input device, which then submits the stream to Intuition for further processing.

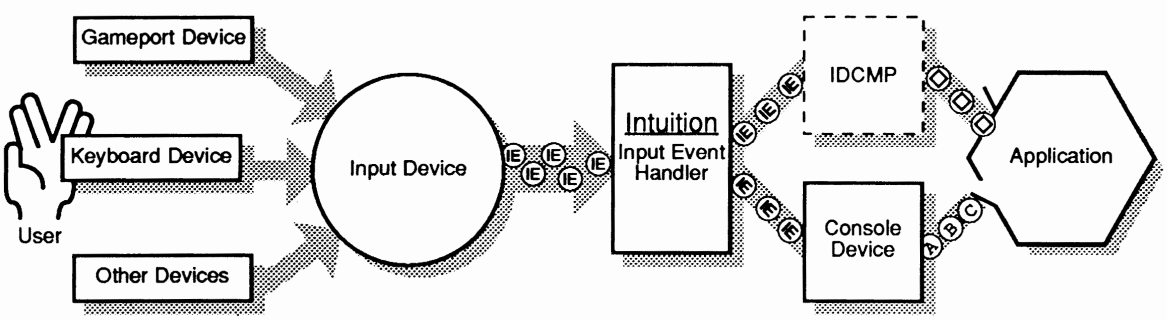


Figure 9-1: Amiga Input Block Diagram

The application program can receive its input from Intuition or the Console device. The application may choose to listen to neither, one or both of these input sources.

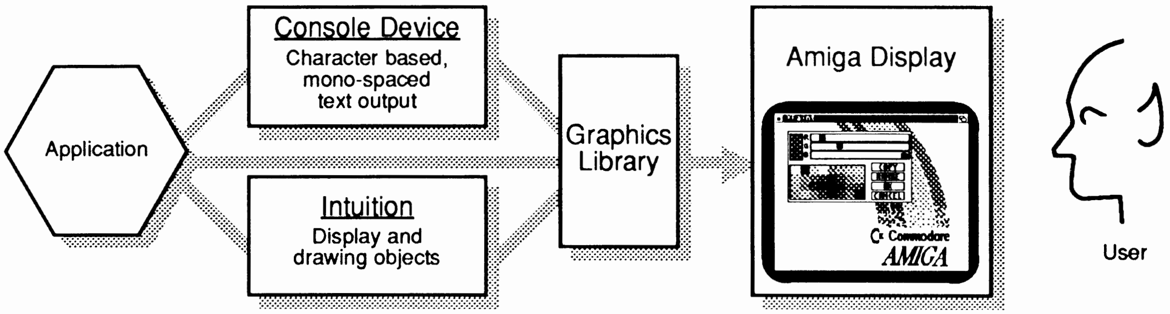


Figure 9-2: Amiga Output Block Diagram

An application's display output can go through the high level interfaces of the console device or through the Intuition library. Additionally, display output may be sent directly to the graphics library. Notice that both the Console and Intuition call the graphics library to render to the display.

Intuition Input

The Amiga has an input device to monitor all input activity. The input activity nominally includes keyboard and mouse events, but which can be extended to include other types of input signals. When the user moves the mouse, presses a mouse button or types on the keyboard, the input device detects the activity from the specific device, and constructs an **InputEvent**.

An **InputEvent** is a message describing a single event, such as the transition of a key on the keyboard from up to down.

The input device then passes the input events down a prioritized chain of input handlers, which are routines in memory that process the input events. The sequence of input events passing through this chain of input handlers is known as the *input stream.* Any handler linked into this chain can monitor and modify the event stream.

Each input handler may block (consume) events, allow events to pass through to the next handler in the chain or add new events to the sequence.

Other devices and programs can add input events to the input stream by sending messages to the input device. For instance, AmigaDOS is able to generate an input event whenever a disk is inserted or removed.

See the "Input Device" chapter of the *Amiga ROM Kernel Reference Manual: Devices* for more information on the Input device.

INTUITION AS AN INPUT HANDLER

Intuition is an input handler linked into the input stream, and it monitors and modifies events that it receives. The input arrives at Intuition as a single stream of events. These events are filtered, altered, and enhanced by Intuition, then dispatched to windows as appropriate, or passed down to input handlers lower in the chain. If the active window has a console attached to it, then it can receive the input events that are still left in the stream, which can include some events that Intuition played a role in forming.

Many kinds of input event undergo little conversion by Intuition. For instance, raw keyboard events are not modified by Intuition (with the exception of a few keystrokes that have special meaning). Other events may produce differing results based on Intuition's view of the system. For example, when the mouse select button is pressed, the event may become a gadget down-press event, a window activation event, or it may remain a simple button press, depending on the mouse position and the arrangement of windows and screens. Still other events are consumed by Intuition, and the application is not directly notified. An example would be when the select button is pressed over a system gadget.

Intuition is also the originator of certain kinds of events. For example, a window-refreshing event is generated when Intuition discovers that part of a window is in need of redrawing. This might have resulted indirectly from some other input (for example, the user might have dragged a window), but not necessarily (the refresh might have been necessitated by a program bringing a window to the front).

RECEIVING INPUT EVENTS FROM INTUITION

There are two channels through which a window can receive events destined for it. The usual way is for the application to ask Intuition to send it messages which are based on the input event that Intuition has processed. These messages, called **IntuiMessages**, are standard Amiga Exec messages, and are sent to a port called an Intuition Direct Communications Message Port, or IDCMP. Every window may have an IDCMP associated with it (pointed to by **Window.UserPort**).

There are many classes of **IntuiMessages**, and the application can control which classes of events are routed to its window's port by setting the appropriate IDCMP flags. When Intuition has an event to send, but the window does not have the corresponding IDCMP flag set, the event is generally passed along to the next input handler in the chain. One input handler that resides below Intuition's is the console device's handler. If your application's window has a console attached to it, the console device will generally convert events it receives into console code sequences, and send those to your console. In this manner, you can hear these events.

Because **IntuiMessages** and the IDCMP are the primary way in which applications receive communication from Intuition, discussions elsewhere in the manual frequently refer to events from Intuition as messages, **IntuiMessages**, or IDCMP messages. However, most of the information sent as **IntuiMessages** is also available through the console device, though that option is used less often. Elsewhere in this chapter, you can learn how getting your events through the console differs from getting them through your IDCMP.

Whichever way an application chooses to get its messages, it is frequently designed to be event-driven. That is to say, after some amount of initialization, the application will go into a state where it is waiting for some event to happen. This event could be an input event, or some other kind of event. Based on the event received, the application would take appropriate action, and return to its waiting state.

IDCMP EVENTS AND THE INPUT FOCUS

Although at any given time many applications may be waiting for input, in most cases only the active application (the one with the currently active window) will receive IDCMP messages.

Since the IDCMP messages are, in general, directed to a single window, this window is said to have the *input focus*—the input from a variety of sources is focused on this single location.

The active window is generally selected by the user, although it is possible for applications to change the active window. See the "Intuition Windows" chapter for information on selecting or setting the active window. Be aware that changing the active window will change the input focus. Usually this change is performed following user action—the user selects a window with the mouse, or activates a new application. Changes to the input focus without user control, such as activating another window while the user is working in an application, may confuse the user. Perform such changes with great care.

Not all events are sent only to the active IDCMP. Some events, such as "disk inserted," may be useful to many programs, so Intuition translates these events into separate messages, one for each application.

Intuition Output

Visual program output, the information written to the display, is sent through one of three channels.

- Imagery may be sent to the graphics library primitives. Graphics library includes functions for line drawing, area fill, specialized animation and output of text. See the graphics library chapters "Graphics Primitives", "Graphics Library and Text" and "Graphics Sprites, Bobs and Animation" for more information on these functions.
- Use the Intuition library support functions for rendering text, graphical imagery, and line drawing. These provide some of the same functions as the graphics library routines, but the Intuition functions perform more of the detail work for you. See the chapter "Intuition Images, Line Drawing and Text" for more information on Intuition rendering functions. Also see, of course, the chapters on screens, windows, gadgets menus and requesters for information on managing the display.
- Output character-based data via the console device. The console device is discussed in the next section.

Console Device I/O

A program receives its input stream either directly from Intuition or via another mechanism known as the console device.

The console device may be used both as a source for input and as a mechanism for output. Often, it is convenient to use only the console device for input and output. In particular, character-based programs can open the console and use it for all I/O without worrying about windows, bitmaps, or message ports.

The console device gives the program "cooked" input data, including key code conversions to ASCII and conversions of Intuition generated events, such as IDCMP_CLOSEWINDOW, to ANSI escape sequences.

The console device output provides features such as automatic line wrapping and scrolling. If an application just wants to output text, it may choose to use the console device, which provides formatted text with little fuss.

If the application is not character-based, it may be better for the it to use an IDCMP for input and render graphics and text directly through Intuition and the graphics library primitives.

If necessary, it is possible to open both the console device and an IDCMP for input. Such a program might need ASCII input, formatted output and the IDCMP verification functions (for example, to verify that it has finished writing to the window before the user can bring up a requester).

For more information on the console device, see the "Console Device" chapter of the *Amiga ROM Kernel Reference Manual: Devices.*

Using the IDCMP

The IDCMP allow the application to receive information directly from Intuition. The program can use the IDCMP to learn about mouse, keyboard and other Intuition events. Also, certain useful Intuition features, most notably the verification functions (described under "IDCMP Flags" below), require that the IDCMP be opened, as this is the only mechanism available for accessing these features.

The IDCMP consists of a pair of message ports, which may be allocated and initialized by Intuition at the request of the program. Alternately, the application may choose to manage part of the allocation, such that one port is supplied by the application and one port is supplied by Intuition. These ports are standard Exec message ports, used to allow interprocess communications in the Amiga multitasking environment. To learn more about message ports and message passing, see the chapter "Exec Messages and Ports".

The IDCMP is always associated with a window, it is not possible to have an IDCMP without an open window. The IDCMP is made up of several fields in the **Window** structure:

- **IDCMPFlags** stores the IDCMP flags currently set for this port. This field should never be directly set by the application; use the function **ModifyIDCMP()** or set them when the window is opened instead.
- **UserPort** is a pointer to the standard Exec message port where the application receives input event messages from Intuition
- **WindowPort** is a pointer to the reply message port used by Intuition. The messages sent by Intuition are set up such that **ReplyMsg()** will return them to this port.

To open these ports automatically, set at least one of the IDCMP flags in the **OpenWindowTagList()** call. To free these ports later in the program, call the function **ModifyIDCMP()** with NULL for the IDCMP flags or simply close the window.

> *Don't Reply Any Messages After the IDCMP is Freed.* If an IDCMP is freed, either by calling **ModifyIDCMP()** or by closing the window, Intuition will reclaim and deallocate all messages waiting at that port without waiting for a **ReplyMsg()**. If the program attempts to **ReplyMsg()** to an **IntuiMessages** after the IDCMP is closed, the system will probably crash.

If the IDCMP flags are NULL when the window is opened, no ports will be allocated when the window is created. To have Intuition allocate these ports later, call the function **ModifyIDCMP()** with any of the IDCMP flags set. (Starting in V37, **ModifyIDCMP()** returns NULL if it was unable to create the necessary message ports. Do not check the return code under V36 or earlier.)

Once the IDCMP is opened, with the ports allocated, the program can receive many types of information directly from Intuition, based on the IDCMP flags that are set.

The IDCMP allows the application to receive only the events that it considers important. The program can, for instance, choose to learn about gadget events but may not want to learn about other mouse or keyboard events. This is done by providing a "filter" or "mask" value for the IDCMP which tells Intuition which events it should send to this specific port. Only messages with a type matching one of the flags set in the **Window** structure's **IDCMPFlags** field will be sent to this port. These values may be set at creation time, or modified by calling the function **ModifyIDCMP()**.

Messages sent to the IDCMP are instances of the structure **IntuiMessage**. This is an extended form of the Exec **Message** structure which allows Intuition to send user interface specific information to the application. The **IntuiMessage** structure is discussed at length below.

After the application opens an IDCMP, it must monitor the port for messages. At a minimum, this involves removing all messages from the port and replying to them. An event loop which processes messages arriving at the IDCMP is discussed below.

STANDARD INTUIMESSAGE EVENT LOOP

The application should handle events quickly. Any delay in this handling will make the user interface appear sluggish to the user. Additionally, certain events such as IDCMP_SIZEVERIFY may time-out if the application does not respond to them quickly (this is to help prevent system deadlocks). The action taken by Intuition when an event times-out may not match the action desired by the program. When IDCMP_SIZEVERIFY times out, the window sizing operation is cancelled by Intuition.

Code should be able to handle the case where there are multiple events waiting at the port. When events are being generated quickly, Intuition may post many events to the IDCMP before the application regains control. This can happen regardless of how fast the application processes the messages waiting at the port. Since messages queue up but signals do not, the application may not see a signal for each message posted. Because of these facts, the code should remove all the messages waiting at the port, regardless of the number, each time **Wait()** returns.

Code should also be able to handle the case where the signal is set but no events are waiting at the port. This could happen if a new message arrives at the IDCMP while an application is still processing the previous message. Since applications typically process all queued messages before returning to **Wait()**, the second message gets handled with the signal bit still set. The subsequent call to **Wait()** will return immediately even though no message is present. These cases should be quietly ignored.

EVENT LOOP EXAMPLE

This example shows how to receive Intuition events. It reports on a variety of events: close window, keyboard, disk insertion and removal, select button up and down and menu button up and down. Note that the menu button events will only be received by the program if the **WA_RMBTrap** attribute is set for the window.

```
;/* eventloop.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 eventloop.c
Blink FROM LIB:c.o,eventloop.o TO eventloop LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** eventloop.c - standard technique to handle IntuiMessages from an IDCMP.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* our function prototypes */
BOOL handleIDCMP(struct Window *win, BOOL done);

struct Library *IntuitionBase = NULL;


/*
** main routine.
** Open required library and window, then process the events from the
** window.  Free all resources when done.
*/
VOID main(int argc, char **argv)
{
ULONG signals;
UBYTE done;
struct Window *win;

IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library",37);
if (IntuitionBase != NULL)
    {
    if (win = OpenWindowTags(NULL,
                        WA_Title,       "Press Keys and Mouse in this Window",
                        WA_Width,       500,
                        WA_Height,      50,
                        WA_Activate,    TRUE,
                        WA_CloseGadget, TRUE,
                        WA_RMBTrap,     TRUE,
                        WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_VANILLAKEY |
                            IDCMP_RAWKEY | IDCMP_DISKINSERTED |
                            IDCMP_DISKREMOVED | IDCMP_MOUSEBUTTONS,
                        TAG_END))
        {
        done = FALSE;

        /* perform this loop until the message handling routine signals
        ** that we are done.
        **
        ** When the Wait() returns, check which signal hit and process
        ** the correct port.  There is only one port here, so the test
        ** could be eliminated.  If multiple ports were being watched,
        ** the test would become:
```

```
**
**    signals = Wait( (1L << win1->UserPort->mp_SigBit) |
**                    (1L << win2->UserPort->mp_SigBit) |
**                    (1L << win3->UserPort->mp_SigBit))
**    if (signals & (1L << win1->UserPort->mp_SigBit))
**        done = handleWin1IDCMP(win1,done);
**    else if (signals & (1L << win2->UserPort->mp_SigBit))
**        done = handleWin2IDCMP(win2,done);
**    else if (signals & (1L << win3->UserPort->mp_SigBit))
**        done = handleWin3IDCMP(win3,done);
**
** Note that these could all call the same routine with different
** window pointers (if the handling was identical).
**
** handleIDCMP() should remove all of the messages from the port.
*/
while (!done)
    {
    signals = Wait(1L << win->UserPort->mp_SigBit);
    if (signals & (1L << win->UserPort->mp_SigBit))
        done = handleIDCMP(win,done);
    };
CloseWindow(win);
}
    CloseLibrary(IntuitionBase);
    }
}


/*
** handleIDCMP() - handle all of the messages from an IDCMP.
*/
BOOL handleIDCMP(struct Window *win, BOOL done)
{
struct IntuiMessage *message;
USHORT code;
SHORT mousex, mousey;
ULONG class;

/* Remove all of the messages from the port by calling GetMsg()
** until it returns NULL.
**
** The code should be able to handle three cases:
**
** 1.  No messages waiting at the port, and the first call to GetMsg()
** returns NULL.  In this case the code should do nothing.
**
** 2.  A single message waiting.  The code should remove the message,
** processes it, and finish.
**
** 3.  Multiple messages waiting.  The code should process each waiting
** message, and finish.
*/
while (NULL != (message = (struct IntuiMessage *)GetMsg(win->UserPort)))
    {
    /* It is often convenient to copy the data out of the message.
    ** In many cases, this lets the application reply to the message
    ** quickly.  Copying the data is not required, if the code does
    ** not reply to the message until the end of the loop, then
    ** it may directly reference the message information anywhere
    ** before the reply.
    */
    class  = message->Class;
    code   = message->Code;
    mousex = message->MouseX;
    mousey = message->MouseY;

    /* The loop should reply as soon as possible.  Note that the code
    ** may not reference data in the message after replying to the
    ** message.  Thus, the application should not reply to the message
    ** until it is done referencing information in it.
    **
    ** Be sure to reply to every message received with GetMsg().
    */
    ReplyMsg((struct Message *)message);
```

```
    /* The class contains the IDCMP type of the message. */
    switch (class)
        {
        case IDCMP_CLOSEWINDOW:
            done = TRUE;
            break;
        case IDCMP_VANILLAKEY:
            printf("IDCMP_VANILLAKEY (%lc)\n",code);
            break;
        case IDCMP_RAWKEY:
            printf("IDCMP_RAWKEY\n");
            break;
        case IDCMP_DISKINSERTED:
            printf("IDCMP_DISKINSERTED\n");
            break;
        case IDCMP_DISKREMOVED:
            printf("IDCMP_DISKREMOVED\n");
            break;
        case IDCMP_MOUSEBUTTONS:
            /* the code often contains useful data, such as the ASCII
            ** value (for IDCMP_VANILLAKEY), or the type of button
            ** event here.
            */
            switch (code)
                {
                case SELECTUP:
                    printf("SELECTUP at %d,%d\n",mousex,mousey);
                    break;
                case SELECTDOWN:
                    printf("SELECTDOWN at %d,%d\n",mousex,mousey);
                    break;
                case MENUUP:
                    printf("MENUUP\n");
                    break;
                case MENUDOWN:
                    printf("MENUDOWN\n");
                    break;
                default:
                    printf("UNKNOWN CODE\n");
                    break;
                }
            break;
        default:
            printf("Unknown IDCMP message\n");
            break;
        }
    }
return(done);
}
```

SETTING UP A CUSTOM USER PORT

An application can use its own message port for the IDCMP instead of the one set up by Intuition, although some care is required.

As described earlier, IDCMP communication takes place through a pair of Exec message ports attached to a window: the **UserPort** and the **WindowPort**. The **UserPort** is the port where the application receives IDCMP messages from Intuition. The **WindowPort** is the reply port where Intuition receives replies from the application (via the **ReplyMsg()** function).

In the simplest case, Intuition allocates (and deallocates) both of these ports when the program opens a window with non-NULL IDCMP flags. Intuition will also allocate these ports if the application calls **ModifyIDCMP()** with non-NULL flags for a window that has NULL IDCMP flags. These port variables will be set to NULL if there is no message port allocated, otherwise they will contain a pointer to a message port.

If the **WindowPort** is not already opened when either **OpenWindow()** or **ModifyIDCMP()** is called, it will be allocated and initialized.

The **UserPort** is checked separately to see whether it is already opened.

When Intuition initializes the **UserPort**, it also allocates a signal bit with a call to **AllocSignal()**. Since the application makes the call to **OpenWindowTagList()** or **ModifyIDCMP()**, this signal bit is valid for the application's task. The address of the application's task is saved in the **SigTask** variable of the message port.

The program may choose to supply its own **UserPort**. This might be done in an environment where the program is using several windows and would prefer to monitor the input using only one message port. This is done by with the following procedure:

1. Create a port for the IDCMP by calling either the Exec function **CreateMsgPort()** or the *amiga.lib* function **CreatePort()**, both of which return a pointer to a port. (**CreateMsgPort()** is a new Exec function in V36 and can therefore only be used on systems running Release 2 or a later version of the OS.)

2. Open the windows with no IDCMP flags set. This will prevent Intuition from allocating a port for this window.

3. Place a pointer to the port created in step 1 into the **UserPort** field of the **Window** structure.

4. Call **ModifyIDCMP()** to set the desired IDCMP flags for the port. Intuition will use the port supplied with the window.

   Be Careful with Shared IDCMP Ports. If the application is sharing an IDCMP among several windows, it must be very careful not to call **ModifyIDCMP(window,NULL)** for any of the windows that are using the shared port, as this will free the port and the signal bit.

5. When an application decides to close a window that has a shared IDCMP, there may be messages waiting at the port for any of the windows including the window being closed. It is essential that messages destined for a given window be removed and replied to before that window is closed.

   CloseWindowSafely(), listed in the next example, performs proper message cleanup before closing such a window. It also sets the window's **UserPort** to NULL so that Intuition knows not to delete the port, which should be done by the application in this case. It is incorrect (and dangerous) to simply call **CloseWindow()** on a window that has a shared IDCMP.

   Note that **CloseWindowSafely()** assumes that the window has a **UserPort**.

6. After all windows have been closed, and the port has been removed from each, delete the port that was created in step 1. Use the *amiga.lib* function **DeletePort()** (if BCreatePort() **was used) or the Exec function DeleteMsgPort() (if CreateMsgPort() was used).**

Closing a Window with a Shared IDCMP

As promised in the last section, this example shows the **CloseWindowSafely()** function. Use this function to close any windows that share an IDCMP port with another window.

```
/* CloseWindowSafely
**
** these functions close an Intuition window that shares a port with other
** Intuition windows.
**
** We are careful to set the UserPort to NULL before closing, and to free
** any messages that it might have been sent.
*/
#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/ports.h"
#include "intuition/intuition.h"

/*
** function to remove and reply all IntuiMessages on a port that have been
** sent to a particular window (note that we don't rely on the ln_Succ pointer
** of a message after we have replied it)
*/
VOID StripIntuiMessages(struct MsgPort *mp, struct Window *win)
{
struct IntuiMessage *msg;
struct Node *succ;

msg = (struct IntuiMessage *)mp->mp_MsgList.lh_Head;

while (succ = msg->ExecMessage.mn_Node.ln_Succ)
      {
      if (msg->IDCMPWindow == win)
            {
            /* Intuition is about to free this message.
            ** Make sure that we have politely sent it back.
            */
            Remove(msg);

            ReplyMsg(msg);
            }
      msg = (struct IntuiMessage *)succ;
      }
}

/*
** Entry point to CloseWindowSafely()
** Strip all IntuiMessages from an IDCMP which are waiting for a specific
** window.  When the messages are gone, set the UserPort of the window to NULL
** and call ModifyIDCMP(win,0).  This will free the Intuition arts of the
** IDMCMP and trun off message to this port without changing the original
** UserPort (which may be in use by other windows).
*/
VOID CloseWindowSafely(struct Window *win)
{
/* we forbid here to keep out of race conditions with Intuition */
Forbid();

/* send back any messages for this window  that have not yet been processed */
StripIntuiMessages(win->UserPort, win);

/* clear UserPort so Intuition will not free it */
win->UserPort = NULL;

/* tell Intuition to stop sending more messages */
ModifyIDCMP(win, 0L);

/* turn multitasking back on */
Permit();

/* Now it's safe to really close the window */
CloseWindow(win);
}
```

INTUIMESSAGES

The **IntuiMessage** structure is an Exec **Message** that has been extended to include Intuition specific information. The **ExecMessage** field in the **IntuiMessage** is an actual instance of a **Message** structure and is used by Exec to manage the transmission of the message. The Intuition extensions of the **IntuiMessage** are used to transmit specialized Intuition data to the program.

```
struct IntuiMessage
    {
    struct Message ExecMessage;
    ULONG Class;
    UWORD Code;
    UWORD Qualifier;
    APTR IAddress;
    WORD MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
    };
```

The **IntuiMessage** structure fields are as follows:

ExecMessage

This field is maintained by Exec. It is used for linking the message into the system and broadcasting it to a message port. See the chapter "Exec Messages and Ports" for more information on the **Message** structure and its use.

Class

Class contains the IDCMP type of this specific message. By comparing the **Class** field to the IDCMP flags, the application can determine the type of this message. Each message may only have a single IDCMP type.

Code

Code contains data set by Intuition, such as menu numbers or special code values. The meaning of the **Code** field changes depending on the IDCMP type, or **Class**, of the message. Often the code field will simply contain a copy of the code of the input event which generated this **IntuiMessage**.

For example, when the message is of class IDCMP_RAWKEY, **Code** contains the raw key code generated by the keyboard device. When the message is of class IDCMP_VANILLAKEY, **Code** contains the key mapped ASCII character.

Qualifier

This contains a copy of the **ie_Qualifier** field that is transmitted to Intuition by the input device. This field is useful if your program handles raw key codes, since the **Qualifier** tells the program, for instance, whether or not the Shift key or Ctrl key is currently pressed. Check the *<devices/inputevent.h>* file for the definitions of the qualifier bits.

MouseX and **MouseY**

Every **IntuiMessage** will have the mouse coordinates in these variables. The coordinates can either be expressed as absolute offsets from the upper left corner of the window, or expressed as the amount of change since the last reported positions (delta). If IDCMP_DELTAMOVE is set, then these numbers will represent delta positions from the last position. All messages will have zero in these values except IDCMP_MOUSEMOVE and IDCMP_MOUSEBUTTON events, which will have the correct delta values for the movement. If IDCMP_DELTAMOVE is not set, then these numbers are the actual window offset values.

Seconds and **Micros**

These values are copies of the current system clock, in seconds and microseconds. They are set when Intuition generates the message.

Microseconds (**Micros**) range from zero up to one million minus one. The 32 bits allocated to the **Seconds** variable has enough accuracy to count up to 139 years. Time is measured from Jan 1, 1978.

IAddress

Typically this variable contains the address of some Intuition object, such as a gadget. The type of the object depends on the **Class** of the **IntuiMessage**. Do not assume that the object is of a certain type before checking the **Class** of the object.

The **IAddress** pointer is defined *only* for the following IDCMP **Classes**. Do not attempt to dereference or otherwise interpret the **IAddress** field of any other type of **IntuiMessage**.

IntuiMessage Class	Meaning of IAddress Field
IDCMP_GADGETDOWN	**IAddress** points to the gadget.
IDCMP_GADGETUP	**IAddress** points to the gadget.
IDCMP_RAWKEY	**IAddress** points to the dead-key information.
IDCMP_IDCMPUPDATE	**IAddress** points to a tag item list.
Other classes	No meaning.

In particular, for IDCMP_MOUSEMOVE **IntuiMessages** emanating from GACT_FOLLOWMOUSE gadgets, the **IAddress** field does not point to the gadget. Interpreting the **IAddress** as a gadget pointer and trying to access the gadget's fields before ascertaining that the event is an IDCMP_GADGETUP or IDCMP_GADGETDOWN event is incorrect, and can lead to subtle or serious problems.

(Note that GadTools gadgets do arrange for the **IAddress** to point to the gadget when IDCMP_MOUSEMOVE messages appear).

IDCMPWindow

Contains the address of the window to which this message was sent. If the application is sharing the window's **UserPort** between multiple windows, **IDCMPWindow** allows it to determine which of the windows the message was sent to.

SpecialLink

For system use only.

IDCMP Flags

The application specifies the information it wants Intuition to send to it via the IDCMP by setting IDCMP flags. These may be set either when opening the window or by calling **ModifyIDCMP()**.

The flags set may be viewed as a filter, in that Intuition will only post **IntuiMessages** to an IDCMP if the matching flag is set. Thus, the application will only receive the IDCMP messages whose **Class** matches one of the bits set in the window's IDCMP.

For many of these messages, there is a separation of the act of filtering these messages and causing Intuition to send the messages in the first place. For instance, menu help events may be activated for a window by setting the **WA_MenuHelp** attribute when the window is opened. However, the IDCMP will only receive the messages if the IDCMP_MENUHELP flag is set. If this flag is not set, then the events are passed downstream in the input and may be picked up by the console device.

Mouse Event Message Classes and Flags

IDCMP_MOUSEBUTTONS

Contains reports about mouse button up and down events. The events will be sent to the application only if they are not used internally by Intuition.

The **Code** field contains information on the specific mouse button event this message represents. The **Code** field will be equal to SELECTDOWN, SELECTUP, MENUDOWN, MENUUP, MIDDLEDOWN or MIDDLEUP, depending on the button pressed or released. In general, the select button is the left mouse button, the menu button is the right mouse button and the middle button is an optional third button usually located between the select and menu buttons.

Often, a mouse button event has extra meaning to Intuition, and the application may hear about it through a more specific message, for example a gadget or menu event. Other times, no event is generated at all, such as when the user depth-arranges a screen by clicking on the screen depth gadget. Note that menu button events are normally consumed by Intuition for menu handling. If an application wishes to hear IDCMP_MOUSEBUTTONS events for the menu button, it must set the WA_RMBTrap attribute for its window. See the ''Intuition Windows'' chapter for more information.

IDCMP_MOUSEMOVE

Reports about mouse movements, sent in the form of x and y coordinates relative to the upper left corner of the window. One message will be sent to the application for each ''tick'' of the mouse.

The application can opt to receive IDCMP_MOUSEMOVE events only while certain gadgets are active, or during normal window operation. These events are sent whenever a gadget with GACT_FOLLOWMOUSE gadget is active, or for any window that has the WA_ReportMouse attribute set. This window attribute can be set or cleared by the application at will. See the "Intuition Windows" chapter for full details.

Requesting IDCMP_MOUSEMOVE messages can create a very large volume of messages arriving at the window's IDCMP. Do not request these messages unless the program is prepared to keep up with them. Starting in V36, Intuition limits the number of mouse move events that pile up at your IDCMP.

All IDCMP messages contain a mouse x and y position that can be absolute values or delta values. See IDCMP_DELTAMOVE, below. If the application requires a less frequent reporting of the mouse position, consider using IDCMP_INTUITICKS. While IDCMP_MOUSEMOVE events are generated by changes in the mouse's position, IDCMP_INTUITICKS **IntuiMessages** are based on a timer. Since they contain mouse coordinates, they effectively sample the mouse position. These message come often enough for many applications, but not so frequently as to swamp the application.

The program will not be sent IDCMP_MOUSEMOVE messages while Intuition has the layers of the screen locked (during menu operations and window sizing/dragging). This avoids problems of messages accumulating while the program is blocked, waiting to render into a locked layer.

IDCMP_DELTAMOVE

IDCMP_DELTAMOVE is not a message type, and events will never be sent to the application with this IDCMP identifier. This flag is a modifier, which changes how mouse movements are reported. When this flag is set, mouse movements are sent as delta values rather than as absolute positions.

The deltas are the amount of change of the mouse position from the last reported position. If the mouse does not move, then the delta values will be zero.

This flag works in conjunction with the IDCMP_MOUSEMOVE flag. When IDCMP_DELTAMOVE is set, IDCMP_MOUSEBUTTONS messages will also have relative values, instead of the absolute window position of the mouse.

Delta mouse movements are reported even after the Intuition pointer has reached the limits of the display. That is, if the pointer has reached the edge of the display and the user continues to move the mouse in the same direction, the IDCMP_MOUSEMOVE messages will continue to report changes in the mouse position even though the pointer is no longer moving.

Gadget Event Message Classes and Flags

IDCMP_GADGETDOWN

IDCMP_GADGETDOWN messages are sent when the user selects a gadget that was created with the GACT_IMMEDIATE flag set. The **IntuiMessage** structure's **IAddress** field will contain a pointer to the selected gadget.

IDCMP_GADGETUP

IDCMP_GADGETUP messages are sent when the user selects a gadget that was created with the GACT_RELVERIFY flag set. The **IntuiMessage** structure's **IAddress** field will contain a pointer to the selected gadget.

IDCMP_CLOSEWINDOW

IDCMP_CLOSEWINDOW messages are sent when the user selects the window's close gadget. Intuition does not close the window when the close gadget is selected. Rather, an IDCMP_CLOSEWINDOW message is sent to the window's IDCMP. It is up to the application to clean up and close the window itself. If closing a window means losing some data (perhaps the spreadsheet the user was working on), it would be appropriate for the application to first confirm that the user really meant to close the window.

Menu Event Message Classes and Flags

IDCMP_MENUPICK

This flag indicates that the user has pressed the menu button. If a menu item was selected, the menu number of the menu item can be found in the **Code** field of the **IntuiMessage**. If no item was selected, the **Code** field will be equal to MENUNULL.

IDCMP_MENUVERIFY

This is a special verification mode which allows the program to confirm that it is prepared to handle Intuition rendering, in this case, allowing menus to be drawn in the screen.

This is a special kind of verification, in that any window in the entire screen that has this flag set must respond before the menu operations may proceed. Also, the active window of the screen is allowed to cancel the menu operation. This is unique to IDCMP_MENUVERIFY. Refer to the "Intuition Menus" for a complete description.

Also see the "Verification Functions" section below for more information.

IDCMP_MENUHELP

This message is sent by Intuition when the user selects the Help key while the menu system is activated. If a menu item was selected, the menu number of the menu item can be found in the **Code** field of the **IntuiMessage**. If no item was selected, the **Code** field will be equal to MENUNULL.

These messages will only be sent if the **WA_MenuHelp** attribute is set for the window.

The menu number returned in IDCMP_MENUHELP may specify a position that cannot be generated through normal menu activity. For instance, the menu number may indicate one of the menu headers with no item or sub-item. See the chapter on "Intuition Menus" for more information.

Requester Event Message Classes and Flags

IDCMP_REQSET

Intuition sends an IDCMP_REQSET message to the window each time a requester opens in that window.

IDCMP_REQCLEAR

Intuition sends an IDCMP_REQCLEAR message to the window each time a requester is cleared from that window.

IDCMP_REQVERIFY

Set this flag to allow the application to ensure it is prepared for Intuition to render a requester in the window. With this flag set, Intuition sends the application a message that a requester is pending, and then waits for the application to reply before drawing the requester in the window.

If several requesters open in the window, Intuition asks the application to verify only the first one. After that, Intuition assumes that all output is being held off until all the requesters are gone.

By setting the IDCMP_REQSET and IDCMP_REQCLEAR flags, the application can track how many requesters are open in the window and when the last requester is cleared. Once all of the requesters are cleared from the window, it is safe to write to the window until another IDCMP_REQVERIFY is received.

See the "Verification Functions" section below for more discussion on using this flag.

Window Event Message Classes and Flags

IDCMP_NEWSIZE

Intuition sends this message after the user has resized the window. After receiving this, the program can examine the size variables in the window structure to discover the new size of the window. The message is sent, even if the size of the window did not actually change.

IDCMP_REFRESHWINDOW

This message is sent whenever the window needs refreshing. This flag makes sense only with windows that have a refresh type of WA_SimpleRefresh or WA_SmartRefresh.

As a minimum, the application must call **BeginRefresh()** and **EndRefresh()** for the window after receiving an IDCMP_REFRESHWINDOW event. Create the window with the WA_NoCareRefresh attribute if you do not want to manage these events. See the "Intuition Windows" chapter for details.

Most of the graphics library calls used for display output are compatible with Intuition, with the exception of **ScrollRaster()**. Intuition will not send an IDCMP_REFRESHWINDOW event when damage is caused to a window by **ScrollRaster()**. This may happen in a simple refresh window which is partially obscured by another windowthe region that scrolls out from behind the front window will be damaged, but the window will receive no notification. Check the LAYERREFRESH bit in the **Layer** structure **Flags** field to see if damage did happen as a result of **ScrollRaster()**.

IDCMP_SIZEVERIFY

Set this flag if the program must complete some operation before the user sizes the window. When the user sizes the window, Intuition sends an IDCMP_SIZEVERIFY message to the application and then waits until the program replies before allowing the user to size the window. See the "Verification Functions" section below for some things to consider when using this flag.

IDCMP_ACTIVEWINDOW and **IDCMP_INACTIVEWINDOW**

Set these flags to discover when the window becomes activated or deactivated.

Other Event Message Classes and Flags

IDCMP_VANILLAKEY

IDCMP_VANILLAKEY messages return keyboard events translated into the current default character keymap. The mapped character value is returned in the **Code** field of the **IntuiMessage** structure.

An IDCMP_VANILLAKEY message is sent only if the translation results in a single byte value, therefore the program cannot read the Help or function keys using IDCMP_VANILLAKEY.

Starting with V36, programs using IDCMP_VANILLAKEY which also require the additional information of special keys, such as the Help key and the function keys, may set both IDCMP_VANILLAKEY and IDCMP_RAWKEY. When this combination is used, all keypresses that map to single character values will be returned as IDCMP_VANILLAKEY events; all other keyboard events will be sent as IDCMP_RAWKEY messages. Note that IDCMP_VANILLAKEY processing uses all of the key-up events, so the application will only receive key-down events in the IDCMP_RAWKEY format.

IDCMP_RAWKEY

IDCMP_RAWKEY messages give the raw keycodes from the keyboard. The numeric value of the keycode is sent in the **Code** field. Separate codes are returned for key down and key up. Qualifier codes, such as Shift or Alt and whether this key is a repeat, may be found in the **Qualifier** field of the message.

In general, the application should not assume any correspondence between the keycode and the key value. Character positions on the keyboard change from country to country, and the application should respect the keymap set by the user.

Programs using IDCMP_RAWKEY messages should perform their own key mapping by calling the console.device function **RawKeyConvert()**, or the keymap.library function **MapRawKey()**. (The latter is a bit more convenient, but is only available under V36 and higher). The Autodoc for the **MapRawKey()** function shows how you can process so-called *dead keys*. A dead key is a key combination that has no immediate effect, but instead modifies a subsequent keystroke. For example, on the default keymap, Alt-F is a dead key for the acute accent mark. The sequence of Alt-F followed by the E key yields an é with an acute accent.

For an example of key mapping using the **RawKeyConvert()** call, see the rawkey.c example in the ''Intuition Mouse and Keyboard'' chapter.

The application can assume that certain keys will always return the same raw keycode, these keys do not have to be mapped. In general these keys are in the high part of the keymap, above hex 40, and includes all non-alphanumeric keys. The fixed keys include the function keys, backspace, delete, help and cursor keys.

IDCMP_NEWPREFS

IDCMP_NEWPREFS messages are sent when the system Preferences are changed by a call to **SetPrefs()**. The program can learn of these changes by setting this flag.

After receiving a message of class IDCMP_NEWPREFS, the application should call **GetPrefs()** to obtain a copy of the new Preferences.

Under the new Preferences scheme used in Release 2 and later versions of the OS, an IDCMP_NEWPREFS message will not always be sent when the user changes a Preferences setting. Only Preferences values available under V34, i.e., those that can be modified by a call to **SetPrefs()**, will cause an IDCMP_NEWPREFS message to be sent. New Preferences items such as overscan or font settings rely on filesystem notification for monitoring changes. See the chapter on ''Preferences'' for more information.

This message type is broadcast to all IDCMP that have this flag set, not just to the active window. If the application has this flag set, it should be prepared to handle the event even if it is not active.

IDCMP_DISKINSERTED and **IDCMP_DISKREMOVED**

When the user inserts or removes a floppy disk from any drive, Intuition will send one of these message types.

This message type is broadcast to all IDCMP that have this flag set, not just to the active window. If the application has this flag set, it should be prepared to handle the event even if it is not active.

IDCMP_INTUITICKS

Intuition sends these messages to the active window based on an internal timer which ''ticks'' roughly ten times a second. This provides the application with simple timer events from Intuition.

Intuition does not allow IDCMP_INTUITICKS events to accumulate at a port. After an IDCMP_INTUITICKS message has been sent to a port, Intuition will not send another until the application replies to the first. This means that an application that has not been able to service the IDCMP for an extended period can expect at most one IDCMP_INTUITICKS message to be waiting at the port.

These events are to be used as ''prods'', and not as time counters. Do not rely on the timing accuracy of the event, or on the exact frequency at which they appear. Remember, IDCMP_INTUITICKS will only be sent to the active window. If the user selects another window, the events will no longer be received at the first window.

IDCMP_IDCMPUPDATE
Used for notification from Boopsi custom gadgets. See the chapter on ''BOOPSI'' for more information. The **IAddress** field contains a pointer to a tag item list. Tag lists are described in the chapter ''Utility Library''.

IDCMP_CHANGEWINDOW
This message provides the window with notification of any change in the size or position of a window.

There are two other message classes reserved for system use:

IDCMP_WBENCHMESSAGE
Special messages for Workbench, system use only.

IDCMP_LONELYMESSAGE
For internal tracking by Intuition, system use only.

VERIFICATION FUNCTIONS

IDCMP_SIZEVERIFY, IDCMP_REQVERIFY and IDCMP_MENUVERIFY are exceptional in that Intuition sends an **IntuiMessage** to the application and then waits for the application to reply before Intuition proceeds. The application replies by calling the Exec function **ReplyMsg**().

The implication is that the user requested some operation but the operation will not happen immediately and, in fact, will not happen at all until the application says it is safe. Because this delay can be frustrating and intimidating, the program should strive to make the delay as short as possible. An application should always reply to a verification message as soon as possible.

These problems may be overcome by setting up a separate task to monitor the IDCMP and respond to incoming IntuiMessages immediately. This is recommended where there is heavy traffic through the IDCMP, which occurs when many IDCMP flags are set. Monitoring with a separate task may not be appropriate if the main program must synchronize with the event before it can respond to the message.

In previous versions of the operating system, it was not safe to leave any of the VERIFY functions enabled at a time when the task is unable to respond for a long period. This restriction included calls to AmigaDOS directly (with **Open**(), for example), or indirectly (with **OpenLibrary**(), for a disk based library, for example), when a VERIFY function was active. This was because there are many cases where AmigaDOS will put up a requester prompting the user for input, and Intuition may end up waiting for the application to reply to the VERIFY message, while the application waits for the AmigaDOS call to finish. Prior to Release 2, this deadlock would freeze the Amiga.

Beginning with V36, Intuition will no longer wait forever for the application to respond to the verify messages. These messages will now time-out; that is, if the application does not respond within a set period, Intuition will act as if it had. Even in this case, though, the machine will appear to be locked up until the time-out occurs.

The application should use **ModifyIDCMP()** to turn off all VERIFY messages before calling AmigaDOS, or functions that may call AmigaDOS.

If the application sets up a separate task to monitor the IDCMP, and the task monitoring the IDCMP does not call AmigaDOS functions, and if the monitor task will always be able to reply to the VERIFY message without any help from the other task, then the above warning does not apply.

For additional information, see the IDCMP_MENUVERIFY discussion in the ''Intuition Menus'' chapter, the IDCMP_REQVERIFY discussion in the ''Intuition Requesters and Alerts'' chapter and the IDCMP_SIZEVERIFY discussion in the ''Intuition Windows'' chapter.

This message type is broadcast to all IDCMP on the screen that have this flag set, not just to the active window. If the application has this flag set, it should be prepared to handle the event even if it is not active.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of input and output under Intuition. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 9-1: Functions for Intuition Input and Output

Function	Description
ModifyIDCMP()	Change the message filter for an IDCMP. Starting in V37, this function has a return value.

Chapter 10
INTUITION MOUSE AND KEYBOARD

In the Intuition system, the mouse is the normal method of making selections and the keyboard is used for entering character data. This section describes how users employ the mouse to interact with the system and how to arrange for a program to use the mouse. It also describes the use of the keyboard, both as a character input device and as an alternate method of controlling the mouse pointer.

The Mouse

The Amiga mouse is a small, hand-held input device connected to the Amiga by a flexible cable. The user can input horizontal and vertical coordinates with the mouse by sliding it around on a smooth surface. This movement causes the repositioning of a pointer on the display; whenever the mouse is moved the pointer moves, and in the same direction.

The mouse also provides two or three input keys, called mouse buttons, that allow the user to input information to the computer. The basic activities the user can perform with the mouse are shown below.

Action	Explanation
Moving the Mouse	Sliding the body of the mouse over a surface, such as a desk top.
Pressing a button	Pushing down a mouse button (which is released at some later time).
Clicking a button	Quickly pressing and releasing a mouse button.
Double clicking a button	Clicking a button twice in a short period of time.
Dragging	Pressing a button and moving the mouse while the button is held down. The drag operation is completed by releasing the button.

Table 10-1: Mouse Activities

The action associated with mouse button presses can occur when the button is first pressed, or while the button is held down, or when the button is released. As an example of this, consider the drag gadget of a window. When the select button of the mouse is first pressed an outline representing the window frame is drawn. While the button is held down the outline remains, and it moves with the pointer as the mouse is moved. When the button is released, the outline is erased and the window takes its new position.

INTUITION'S USE OF MOUSE EVENTS

When the mouse is moved or its buttons are pressed, the system generates input events that represent the actions. The input events are taken from the input chain by Intuition when the active window requires the events. Note that only input for a specific window will be affected by changes in that window's IDCMP flags.

Most events generated by the user with the mouse are used by Intuition.

As the user moves the mouse, Intuition changes the position of its pointer. The Intuition pointer moves around the entire video display, mimicking the user's movement of the mouse. The user points at an object by positioning the *hot spot* of the pointer over the object. The hot spot is the active part of the pointer image; the hot spot for Intuition's default pointer is the pixel at the tip of the arrow.

After pointing to an object, the user can perform some action on that object by selecting it with one of the mouse buttons. These can include any of the actions specified above, such as dragging or double clicking.

The left mouse button is generally used for selection, while the right mouse button is most often used for information transfer. The terms selection and information are intentionally left open to some interpretation, as it is impossible to imagine all the possible uses for the mouse buttons.

The selection/information paradigm can be crafted to cover most interaction between the user and an application. When using the mouse, the application should emphasize this model. It will help the user to understand and remember the mouse control of the application.

Applications that handle mouse button events directly, bypassing the menu and gadget systems, should use the same selection/information model used by Intuition.

Select Button

When the user presses the left, or select button, Intuition examines the state of the system and the position of the pointer. This information is used to decide whether or not the user is trying to select some object, operation, or option. For example, the user positions the pointer over a gadget and then presses the left button to select that gadget. Alternatively, the user can position the pointer over a window and press the select button to activate the window. The pointer is said to be over an object when the pointer's hot spot is positioned within the selection region of the object.

A number of other common techniques involving the select button are available. They include:

Drag Select

Multiple objects or an extended area may be selected by dragging the mouse over a range with the select button held down. For instance, in Release 2, multiple icons can be selected in a Workbench window by pressing the select button while the pointer is over the background of the window (not an icon or a system gadget) and then moving the mouse with the select button held down. A selection rectangle will be displayed and all icons within the rectangle will be selected. Similarly, the user may highlight blocks of text in a console window by pressing the select button over the first desired character and dragging the mouse to the last desired character while holding the button down.

Multi-Select or Shift Select

Another way to select multiple objects or an extended area is through the shift select technique. First, select the first member of the group of objects in the normal way. Additional objects can be added to the group by holding down the Shift key while the select button is pressed. This technique works with Workbench icons, where icons may be added one-at-a-time to the list of selected icons; and with text in a console window, where the selected text is extended to include the new position. Note that text need not operate this way, and the application may allow multiple discrete blocks to be selected at any given time.

Cancel Drag Operation

Both drag select and the dragging of individual objects may often be canceled by pressing the right mouse button before completing the drag operation (before releasing the select button). Examples of this include window dragging and sizing, and positioning of Workbench icons.

Menu Button

The right mouse button is used to initiate and control information gathering processes. Intuition uses this button most often for menu operations.

For most active windows, pressing the menu button will display the window's menu bar at the top of the screen. Dragging the mouse with the menu button depressed allows the user to browse through the available menus. Releasing the right mouse button over a menu item will select that item, if it is a valid choice. Additionally, the user can select multiple items by repeatedly pressing the select button while the menu button is held down.

Drag selection is also available in menu operations. When the menu system is activated, and the user has the menu button pressed, the select button may be pressed and the mouse dragged over all items to be selected. This only works if the select button is pressed after the menu button, and all items that the pointer travels over will be selected.

Double clicking the right mouse button can bring up a special requester for extended exchange of information. This requester is called the double-menu requester, because a double click of the menu button is required to reveal it, and because this requester acts like a super menu through which a complex exchange of information can take place. Because the requester is used for the transfer of information, it is appropriate that this mechanism is called up by using the right button.

The programmer should consult the *Amiga User Interface Style Guide* for more information on the standard uses of the mouse and its buttons.

Button activation and mouse movements can be combined to create compound instructions. For example, Intuition combines multiple mouse events when displaying the menu system. While the right button is pressed to reveal the menu items of the active window, the user can move the mouse to position the pointer and display different menu items and sub-items. Additionally, multiple presses of the left button can be used to select more than one option from the menus.

Dragging can have different effects, depending on the object being dragged. Dragging a window by the drag gadget will change the position of the window. Dragging a window by the sizing gadget will change the size of the window. Dragging a range in a Workbench window will select all of the icons in the rectangular range.

MOUSE MESSAGES

Mouse events are broadcast to the application via the IDCMP or the console device. See the "Intuition Input and Output Methods" chapter in this book for information on the IDCMP. See the "Console Device" chapter in the *Amiga ROM Kernel Reference Manual: Devices* for more about the console device.

Simple mouse button activity not associated with any Intuition function will be reported to the window as an IntuiMessage with a **Class** of IDCMP_MOUSEBUTTONS. The **IntuiMessage Code** field will be set to SELECTDOWN, SELECTUP, MIDDLEDOWN, MIDDLEUP, MENUDOWN or MENUUP to specify changes in the state of the left, middle and right buttons, respectively.

Direct select button events will not be received by the program if the select button is pressed while the pointer is positioned over a gadget or other object which uses the button event. For example, select button activity over a gadget is reported with a **Class** of IDCMP_GADGETDOWN or IDCMP_GADGETUP. The gadget is said to have *consumed* the mouse events and *produced* gadget events.

If the menu system is enabled, menu selections appear with a **Class** of IDCMP_MENUPICK. To directly receive menu button events, the application must set the flag WFLG_RMBTRAP for the window either when the window is opened or by changing the flag in a single, atomic operation. See the chapter "Intuition Windows" for more information on the flag WFLG_RMBTRAP.

The program receives mouse position changes in the event **Class** IDCMP_MOUSEMOVE. The **MouseX** and **MouseY** position coordinates describe the position of the mouse relative to the upper left corner of the reference window. These coordinates are always in the resolution of the screen being used, and may represent any pixel position on the screen, even though the hardware sprites can be positioned only on the even numbered pixels of a high resolution screen and on the even numbered rows of an interlaced screen. Enabling IDCMP_MOUSEMOVE messages is discussed below in the section on "The Pointer".

To get mouse movement reported as deltas (amount of change from the last position) instead of as absolute positions, set the IDCMP flag IDCMP_DELTAMOVE. When IDCMP_DELTAMOVE is set, the IDCMP_MOUSEMOVE messages received by the program will have delta values rather than absolute values. Note that IDCMP_DELTAMOVE is simply a flag used to modify the behavior of IDCMP_MOUSEMOVE, and that no messages of class IDCMP_DELTAMOVE are ever sent.

Each window has a queue limit for the number of IDCMP_MOUSEMOVE messages waiting on its IDCMP at any given time. If the number of mouse move messages waiting at the IDCMP is equal to the queue limit, then Intuition will discard additional IDCMP_MOUSEMOVE messages until the application replies to one of the queued mouse move messages. The default queue limit for mouse move messages is five.

Be aware that this may cause some data loss, especially when the application is using IDCMP_DELTAMOVE, as the information contained in the discarded messages is not repeated. When using IDCMP_DELTAMOVE, this could cause the application to lose track of the actual pointer position. The application may wish to change the default mouse queue size if it is unable to reply to messages queued at the IDCMP for an extended period. The mouse queue can be set when the window is opened by using the WA_MouseQueue tag, and may later be modified using the **SetMouseQueue**() call. Note that the actual mouse position is always available to the application through the **Window** structure **MouseX** and **MouseY**.

MOUSE USAGE EXAMPLE

The example program below shows the use of IDCMP_MOUSEBUTTONS, IDCMP_MOUSEMOVE and **DoubleClick()**. **DoubleClick()** is used to test the interval between two times and determine if the interval is within the user specified time for double clicking as set in the Preferences Input editor.

```
BOOL DoubleClick( unsigned long sSeconds, unsigned long sMicros,
    unsigned long cSeconds, unsigned long cMicros );
```

The **sSeconds** and **sMicros** arguments specify a timestamp value describing the start of the double click time interval to be tested. The **cSeconds** and **cMicros** arguments specify a timestamp value describing the end of the double click time interval to be tested.

DoubleClick() returns TRUE if the time interval was short enough to qualify as a double-click. A FALSE return indicates that the time interval between presses took too long. The button presses should be treated as separate events in that case.

```
;/* mousetest.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 mousetest.c
Blink FROM LIB:c.o,mousetest.o TO mousetest LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** mousetest.c - Read position and button events from the mouse.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <devices/inputevent.h>

#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define BUFSIZE 16

/* something to use to track the time between messages
** to test for double-clicks.
*/
typedef struct myTimeVal
    {
    ULONG LeftSeconds;
    ULONG LeftMicros;
    ULONG RightSeconds;
    ULONG RightMicros;
    } MYTIMEVAL;
```

```
/* our function prototypes */
VOID doButtons(struct IntuiMessage *msg, MYTIMEVAL *tv);
VOID process_window(struct Window *win);

struct Library *IntuitionBase;
struct GfxBase       *GfxBase;      /* we need GfxBase->DefaultFont */


/*
** main() -- set-up everything.
*/
VOID main(int argc, char **argv)
{
struct Window *win;
struct Screen *scr;
struct DrawInfo *dr_info;
ULONG width;

/* Open the libraries we will use.  Requires Release 2 (KS V2.04, V37) */
if (IntuitionBase = OpenLibrary("intuition.library",37))
    {
    if (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 37))
        {
        /* Lock the default public screen in order to read its DrawInfo data */
        if (scr = LockPubScreen(NULL))
            {
            if (dr_info = GetScreenDrawInfo(scr))
                {
                /* use wider of space needed for output (18 chars and spaces)
                 * or titlebar text plus room for titlebar gads (approx 18 each)
                 */
                width = max((GfxBase->DefaultFont->tf_XSize * 18),
                            (18 * 2) + TextLength(&scr->RastPort,"MouseTest",9));

                if (win = OpenWindowTags(NULL,
                            WA_Top,    20,
                            WA_Left,   100,
                            WA_InnerWidth,  width,
                            WA_Height, (2 * GfxBase->DefaultFont->tf_YSize) +
                                       scr->WBorTop + scr->Font->ta_YSize + 1 +
                                       scr->WBorBottom,
                            WA_Flags, WFLG_DEPTHGADGET | WFLG_CLOSEGADGET |
                                      WFLG_ACTIVATE    | WFLG_REPORTMOUSE |
                                      WFLG_RMBTRAP     | WFLG_DRAGBAR,
                            WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_RAWKEY |
                                      IDCMP_MOUSEMOVE   | IDCMP_MOUSEBUTTONS,
                            WA_Title, "MouseTest",
                            WA_PubScreen, scr,
                            TAG_END))
                    {
                    printf("Monitors the Mouse:\n");
                    printf("    Move Mouse, Click and DoubleClick in Window\n");

                    SetAPen(win->RPort,dr_info->dri_Pens[TEXTPEN]);
                    SetBPen(win->RPort,dr_info->dri_Pens[BACKGROUNDPEN]);
                    SetDrMd(win->RPort,JAM2);

                    process_window(win);

                    CloseWindow(win);
                    }
                FreeScreenDrawInfo(scr, dr_info);
                }
            UnlockPubScreen(NULL,scr);
            }
        CloseLibrary((struct Library *)GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

```
/*
** process_window() - simple message loop for processing IntuiMessages
*/
VOID process_window(struct Window *win)
{
USHORT done;
struct IntuiMessage *msg;
MYTIMEVAL tv;
UBYTE prt_buff[14];
LONG xText, yText;  /* places to position text in window. */

done = FALSE;
tv.LeftSeconds = 0; /* initial values for testing double-click */
tv.LeftMicros  = 0;
tv.RightSeconds = 0;
tv.RightMicros  = 0;
xText = win->BorderLeft + (win->IFont->tf_XSize * 2);
yText = win->BorderTop + 3 + win->IFont->tf_Baseline;

while (!done)
    {
    Wait((1L<<win->UserPort->mp_SigBit));

    while ((!done) &&
           (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        switch (msg->Class)
            {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            /* NOTE NOTE NOTE:  If the mouse queue backs up a lot, Intuition
            ** will start dropping MOUSEMOVE messages off the end until the
            ** queue is serviced.  This may cause the program to lose some
            ** of the MOUSEMOVE events at the end of the stream.
            **
            ** Look in the window structure if you need the true position
            ** of the mouse pointer at any given time.  Look in the
            ** MOUSEBUTTONS message if you need position when it clicked.
            ** An alternate to this processing would be to set a flag that
            ** a mousemove event arrived, then print the position of the
            ** mouse outside of the "while (GetMsg())" loop.  This allows
            ** a single processing call for many mouse events, which speeds
            ** up processing A LOT!  Something like:
            **
            ** while (GetMsg())
            **     {
            **     if (class == IDCMP_MOUSEMOVE)
            **         mouse_flag = TRUE;
            **     ReplyMsg();   NOTE: copy out all needed fields first !
            **     }
            ** if (mouse_flag)
            **     {
            **     process_mouse_event();
            **     mouse_flag = FALSE;
            **     }
            **
            ** You can also use IDCMP_INTUITICKS for slower paced messages
            ** (all messages have mouse coordinates.)
            */
            case IDCMP_MOUSEMOVE:
                /* Show the current position of the mouse relative to the
                ** upper left hand corner of our window
                */
                Move(win->RPort,xText,yText);
                sprintf(prt_buff, "X%5d Y%5d", msg->MouseX, msg->MouseY);
                Text(win->RPort,prt_buff,13);
                break;
            case IDCMP_MOUSEBUTTONS:
                doButtons(msg,&tv);
                break;
            }
        ReplyMsg((struct Message *)msg);
        }
    }
}
```

```
/*
** Show what mouse buttons where pushed
*/
VOID doButtons(struct IntuiMessage *msg, MYTIMEVAL *tv)
{
/* Yes, qualifiers can apply to the mouse also.  That is how
** we get the shift select on the Workbench.  This shows how
** to see if a specific bit is set within the qualifier
*/
if (msg->Qualifier & (IEQUALIFIER_LSHIFT | IEQUALIFIER_RSHIFT))
    printf("Shift ");

switch (msg->Code)
    {
    case SELECTDOWN:
        printf("Left Button Down at X%ld Y%ld", msg->MouseX, msg->MouseY);
        if(DoubleClick(tv->LeftSeconds, tv->LeftMicros, msg->Seconds, msg->Micros))
            printf(" DoubleClick!");
        else
            {
            tv->LeftSeconds = msg->Seconds;
            tv->LeftMicros  = msg->Micros;
            tv->RightSeconds = 0;
            tv->RightMicros  = 0;
            }
        break;
    case SELECTUP:
        printf("Left Button Up   at X%ld Y%ld", msg->MouseX, msg->MouseY);
        break;
    case MENUDOWN:
        printf("Right Button down at X%ld Y%ld", msg->MouseX, msg->MouseY);
        if(DoubleClick(tv->RightSeconds, tv->RightMicros, msg->Seconds, msg->Micros))
            printf(" DoubleClick!");
        else
            {
            tv->LeftSeconds = 0;
            tv->LeftMicros  = 0;
            tv->RightSeconds = msg->Seconds;
            tv->RightMicros  = msg->Micros;
            }
        break;
    case MENUUP:
        printf("Right Button Up   at X%ld Y%ld", msg->MouseX, msg->MouseY);
        break;
    }
printf("\n");
}
```

The Pointer

The system provides a pointer to allow the user to make selections from menus, choose gadgets, and so on. The user may control the pointer with a mouse, the keyboard cursor keys or some other type of controller. The specific type of controller is not important, as long as the proper types of input events can be generated.

The pointer is associated with the active window and the input focus. The active window controls the pointer imagery and receives the input stream from the mouse. The pointer and mouse may be used to change the input focus by selecting another window.

POINTER POSITION

There are two ways to determine the position of the pointer: by direct examination of variables in the window structure at any time, and by examining messages sent by Intuition which inform the application of pointer movement. The pointer coordinates are relative to the upper left corner of the window and are reported in the resolution of the screen, even though the pointer's visible resolution is always in low-resolution pixels (note that the pointer is actually a sprite).

The **MouseX** and **MouseY** fields of the **Window** structure always contain the current pointer x and y coordinates, whether or not the window is the active one. If the window is a GimmeZeroZero window, the variables **GZZMouseX** and **GZZMouseY** in the **Window** structure contain the position of the mouse relative to the upper left corner of the inner window.

If the window is receiving mouse move messages, it will get a set of x,y coordinates each time the pointer moves. To receive messages about pointer movements, the WFLG_REPORTMOUSE flag must be set in the **Window** structure. This flag can be set when the window is opened. The flag can also be modified after the window is open by calling **ReportMouse()**, however C programmers should avoid this function. **ReportMouse()** has problems due to historic confusion about the ordering of its C language arguments. Do not use **ReportMouse()** unless you are programming in assembler. C programmers should set the flag directly in the **Window** structure using an atomic operation.

Most compilers generate atomic code for operations such as `mywindow->flags |= WFLG_REPORTMOUSE` or `mywindow->flags &= ~WFLG_REPORTMOUSE`. If you are unsure of getting an atomic operation from your compiler, you may wish to do this operation in assembler, or bracket the code with a **Forbid()/Permit()** pair.

After the WFLG_REPORTMOUSE flag is set, whenever the window is active it will be sent an IDCMP_MOUSEMOVE messages each time the pointer position changes. The window must have the IDCMP flag IDCMP_MOUSEMOVE set to receive these messages.

Mouse movements can cause a very large number of messages to be sent to the IDCMP, the application should be prepared to handle them efficiently.

Messages about pointer movements may also be activated by setting the flag GACT_FOLLOWMOUSE in an application gadget structure. When this flag is set in a gadget, changes in the pointer position are reported as long as the gadget is selected by the user. These messages are also sent as IDCMP_MOUSEMOVE messages.

CUSTOM POINTER

An application can set a custom pointer for a window to replace the default pointer. This custom pointer will be displayed whenever the window is the active one.

To place a custom pointer in a window, call **SetPointer()**.

```
void SetPointer( struct Window *window, UWORD *pointer, long height,
                 long width, long xOffset, long yOffset );
```

Set the **window** argument to the address of the window that is to receive this custom pointer definition. The **pointer** argument is the address of the data that defines the custom pointer image. The format of this data is discussed in the next section, "The Sprite Data Structure".

The **height** and **width** specify the dimensions of the pointer sprite. There is no height restriction but the width of the sprite must be less than or equal to 16.

The **xOffset** and **yOffset** are used to offset the top left corner of the hardware sprite imagery from what Intuition regards as the current position of the pointer. Another way of describing this is the offset of the default Intuition pointer hot spot from the top left corner of the sprite.

For instance, by specifying offsets of (0,0), the top left corner of the sprite image will be placed at the pointer position. On the other hand, specifying an **xOffset** of -7 (remember, sprites are 16 pixels wide) will center the sprite over the pointer position. Specifying an **xOffset** of -15 will place the right edge of the sprite will be over the pointer position.

> *Specifying the Hot Spot.* For compatibility, the application must specify that the "hot spot" of the pointer is one pixel to the left of the desired position. Changes to the pointer done by a program must compensate for this. The Preferences Pointer editor correctly handles this situation.

To remove the custom pointer from the window, call **ClearPointer()**.

```
void ClearPointer( struct Window *window );
```

Set the **window** argument to the address of the window that is to have its custom pointer definition cleared. The pointer will be restored to the default Intuition pointer imagery

SetPointer() and **ClearPointer()** take effect immediately if the window is active, otherwise, the change will only be displayed when the window is made active.

The Sprite Data Structure

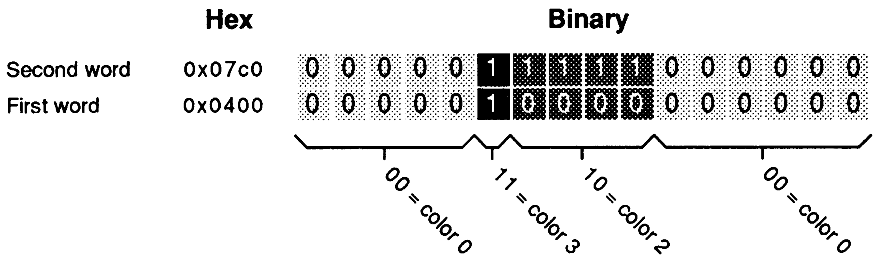
To define the pointer, set up a sprite data structure (sprites are one of the general purpose Amiga graphics structures). The sprite image data must be located in Chip memory, which is memory that can be accessed by the special Amiga hardware chips. Expansion, or Fast memory cannot be addressed by the custom chips. Ensure that data is in Chip memory by using the **AllocMem()** function with the MEMF_CHIP flag, and copying the data to the allocated space. Alternately, use the tools or flags provided by each compiler for this purpose. See the chapter entitled "Exec Memory Allocation", for more information.

A sprite data structure is made up of words of data. In a pointer sprite, the first two words and the last two words are reserved for the system and should be set to zero. All other words contain the sprite image data.

The pointer in the example, a standard busy pointer, is sixteen lines high and sixteen pixels wide. Currently, all sprites are two bit planes deep, with one word of data for each line of each plane. The example sprite image consists of 36 words (2 planes x 18 lines = 36 words). Add to this the four reserved words of control information for a total of 40 words of data. See the example below for the complete data definition.

The sprite data words are combined to determine which color will appear at each pixel position of each row of the sprite. The first two words of image data, 0x0400 and 0x07C0, represent the top line of the sprite. The numbers must be viewed as binary numbers and combined in a bit-wise fashion. The highest bit from each word are combined to form a two bit number representing the color register for the leftmost pixel. The next two bits represent the next pixel in the row, and so on, until the low order bits from each word represent the rightmost pixel in the row.

For example:



Pointer Color Ordering. The first word in a line gives the least significant bit of the color register and the second word gives the most significant bit.

Sprites get their color information from the color registers much like screens do. See the *Amiga Hardware Reference Manual* for more information on the assignment of color registers to sprites. Note that the color number given above is added to a base number to determine the actual hardware color register.

The colors of the Intuition pointer may be changed. The Intuition pointer is always sprite 0. To change the colors of sprite 0, call the graphics library routine **SetRGB4()**.

POINTER EXAMPLE

The program below shows how to set the pointer for a window. In this example, the pointer imagery is changed to a stopwatch symbol which could be used to indicate a busy period.

```
;/* custompointer.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 custompointer.c
Blink FROM LIB:c.o,custompointer.o TO custompointer LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** custompointer.c - Show the use of a custom busy pointer, as well as
** using a requester to block input to a window.
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/libraries.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE
int CXBRK(void)    { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library *IntuitionBase;


UWORD __chip waitPointer[] =
    {
    0x0000, 0x0000,     /* reserved, must be NULL */

    0x0400, 0x07C0,
    0x0000, 0x07C0,
    0x0100, 0x0380,
    0x0000, 0x07E0,
    0x07C0, 0x1FF8,
```

```
    0x1FF0, 0x3FEC,
    0x3FF8, 0x7FDE,
    0x3FF8, 0x7FBE,
    0x7FFC, 0xFF7F,
    0x7EFC, 0xFFFF,
    0x7FFC, 0xFFFF,
    0x3FF8, 0x7FFE,
    0x3FF8, 0x7FFE,
    0x1FF0, 0x3FFC,
    0x07C0, 0x1FF8,
    0x0000, 0x07E0,

    0x0000, 0x0000,     /* reserved, must be NULL */
    };


/*
** The main() routine
*/
VOID main(int argc, char **argv)
{
struct Window *win;
struct Requester null_request;
extern UWORD __chip waitPointer[];

if (IntuitionBase = OpenLibrary("intuition.library",37))
    {
    /* the window is opened as active (WA_Activate) so that the busy
    ** pointer will be visible.  If the window was not active, the
    ** user would have to activate it to see the change in the pointer.
    */
    if (win = OpenWindowTags(NULL,
                             WA_Activate, TRUE,
                             TAG_END))
        {
        /* a NULL requester can be used to block input
        ** in a window without any imagery provided.
        */
        InitRequester(&null_request);

        Delay(50);  /* simulate activity in the program. */

        /* Put up the requester to block user input in the window,
        ** and set the pointer to the busy pointer.
        */
        if (Request(&null_request, win))
            {
            SetPointer(win, waitPointer, 16, 16, -6, 0);

            Delay(100);  /* simulate activity in the program. */

            /* clear the pointer (which resets the window to the default
            ** pointer) and remove the requester.
            */
            ClearPointer(win);
            EndRequest(&null_request, win);
            }

        Delay(100);  /* simulate activity in the program. */

        CloseWindow(win);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

The Keyboard

A program can receive keyboard data through an IDCMP port by setting the IDCMP_RAWKEY flag, the IDCMP_VANILLAKEY flag or both. IDCMP_VANILLAKEY events provide for simple ASCII text and standard control keys like space, return and backspace. IDCMP_RAWKEY events provide a more complex input stream, which the program must process to generate ASCII data. IDCMP_RAWKEY returns all keycodes, both key-up and key-down, including function keys.

> *Keystrokes Are Not Always Paired.* Keystrokes do not always come in key-down/key-up pairs. For example, repeating keys appear as a sequence of key-down messages.

IDCMP_RAWKEY and IDCMP_VANILLAKEY may be set together. When both flags are set in the IDCMP, IDCMP_VANILLAKEY messages will be sent for keystrokes that directly map to a single ASCII value. IDCMP_RAWKEY messages will be sent for key sequences that do not map to simple values, i.e. if a key sequence does not map to an IDCMP_VANILLAKEY message, it will be sent as an IDCMP_RAWKEY message. This allows easy access to mapped characters through IDCMP_VANILLAKEY with control characters returned as IDCMP_RAWKEY. Note that the IDCMP_RAWKEY events will only return the key down events when used with IDCMP_VANILLAKEY.

When Intuition responds to an input event or sequence of events, the application will not receive those events. This happens for system shortcuts (left Amiga + key) if the system shortcut is defined, and for menu shortcuts (right Amiga + key) if the menu shortcut is defined for the active window. If the shortcut is not defined, then the appropriate key event will be sent with the proper Amiga qualifier set.

Key repeat characters have a queue limit which may be set for each window, much like the mouse queue described above. The key repeat queue limit may only be set when the window is opened using the WA_RptQueue tag, there is no function call for modifying the value after the window is open. The default queue limit for key repeat characters is three. This limit causes any IDCMP_RAWKEY, IDCMP_VANILLAKEY or IDCMP_UPDATE message with the IEQUALIFIER_REPEAT bit set to be discarded if the queue is full (IDCMP_UPDATE is discussed in the "BOOPSI" chapter). The queue is said to be full when the number of waiting repeat key messages is equal to the queue limit. Note that the limit is not per character, it is on the total number of key messages with the repeat bit set. Once the limit is reached, no other repeat characters will be posted to the IDCMP until the application replies to one of the outstanding repeat key messages. The repeat queue limit is not as dangerous as the mouse queue limit as only duplicate keystroke information is discarded, where the mouse queue limit discards information that cannot be easily reproduced.

RAWKEY KEYMAPPING EXAMPLE

The following example uses **RawKeyConvert()** to convert the IDCMP_RAWKEY input stream into an ANSI input stream. See the "Console Device" chapter in the *Amiga ROM Kernel Reference Manual: Devices* for more information on **RawKeyConvert()** and the data it returns.

```
;/* rawkey.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 rawkey.c
Blink FROM LIB:c.o,rawkey.o TO rawkey LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** rawkey.c - How to correctly convert from RAWKEY to keymapped ASCII
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
```

```
#include <intuition/intuition.h>
#include <devices/inputevent.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/console_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* our function prototypes */
LONG deadKeyConvert(struct IntuiMessage *msg, UBYTE *kbuffer,
                    LONG kbsize, struct KeyMap *kmap, struct InputEvent *ievent);
VOID print_qualifiers(ULONG qual);
BOOL doKeys(struct IntuiMessage *msg, struct InputEvent *ievent,
                    UBYTE **buffer, ULONG *bufsize);
VOID process_window(struct Window *win, struct InputEvent *ievent,
                    UBYTE **buffer, ULONG *bufsize);

/* A buffer is created for RawKeyConvert() to put its output. BUFSIZE is the size of
** the buffer in bytes.  NOTE that this program starts out with a buffer size of 2.
** This is only to show how the buffer is automatically increased in size  by this
** example!  In an application, start with a much larger buffer and you will probably
** never have to increase its size. 128 bytes or so should do the trick, but always
** be able to change the size if required.
*/
#define BUFSIZE (2)

struct Library *IntuitionBase, *ConsoleDevice;

/* main() - set-up everything used by program. */
VOID main(int argc, char **argv)
{
struct Window *win;
struct IOStdReq ioreq;
struct InputEvent *ievent;
UBYTE *buffer;
ULONG bufsize = BUFSIZE;

if(IntuitionBase = OpenLibrary("intuition.library",37)) {
    /* Open the console device just to do keymapping. (unit -1 means any unit) */
    if (0 == OpenDevice("console.device",-1,(struct IORequest *)&ioreq,0)) {
        ConsoleDevice = (struct Library *)ioreq.io_Device;

        /* Allocate the initial character buffer used by deadKeyConvert() and RawKeyConvert()
        ** for returning translated characters. If the characters generated by these routines
        ** cannot fit into the buffer, the application must pass a larger buffer.  This is
        ** done in this code by freeing the old buffer and allocating a new one.
        */
        if (buffer = AllocMem(bufsize,MEMF_CLEAR)) {
            if (ievent = AllocMem(sizeof(struct InputEvent),MEMF_CLEAR)) {
                if (win = OpenWindowTags(NULL,
                        WA_Width, 300,
                        WA_Height, 50,
                        WA_Flags, WFLG_DEPTHGADGET | WFLG_CLOSEGADGET | WFLG_ACTIVATE,
                        WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_RAWKEY,
                        WA_Title, "Raw Key Example",
                        TAG_END)) {
                    printf("Press keyboard keys to see ASCII conversion from rawkey\n");
                    printf("Unprintable characters will be shown as %c\n\n",0x7f);
                    process_window(win,ievent,&buffer,&bufsize);
                    CloseWindow(win);
                    }
                FreeMem(ievent,sizeof(struct InputEvent));
                }
            /* Buffer can be freed elsewhere in the program so test first. */
            if (buffer != NULL)
                FreeMem(buffer,bufsize);
            }
        CloseDevice((struct IORequest *)&ioreq);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

```
/* Convert RAWKEYs into VANILLAKEYs, also shows special keys like HELP, Cursor Keys,
** FKeys, etc.  It returns:
**   -2 if not a RAWKEY event.
**   -1 if not enough room in the buffer, try again with a bigger buffer.
**   otherwise, returns the number of characters placed in the buffer.
*/
LONG deadKeyConvert(struct IntuiMessage *msg, UBYTE *kbuffer,
    LONG kbsize, struct KeyMap *kmap, struct InputEvent *ievent)
{
if (msg->Class != IDCMP_RAWKEY) return(-2);
ievent->ie_Class = IECLASS_RAWKEY;
ievent->ie_Code = msg->Code;
ievent->ie_Qualifier = msg->Qualifier;
ievent->ie_position.ie_addr = *((APTR*)msg->IAddress);

return(RawKeyConvert(ievent,kbuffer,kbsize,kmap));
}

/* print_qualifiers() - print out the values found in the qualifier bits of
** the message. This will print out all of the qualifier bits set.
*/
VOID print_qualifiers(ULONG qual)
{
printf("Qual:");
if (qual & IEQUALIFIER_LSHIFT)         printf("LShft,");
if (qual & IEQUALIFIER_RSHIFT)         printf("RShft,");
if (qual & IEQUALIFIER_CAPSLOCK)       printf("CapLok,");
if (qual & IEQUALIFIER_CONTROL)        printf("Ctrl,");
if (qual & IEQUALIFIER_LALT)           printf("LAlt,");
if (qual & IEQUALIFIER_RALT)           printf("RAlt,");
if (qual & IEQUALIFIER_LCOMMAND)       printf("LCmd,");
if (qual & IEQUALIFIER_RCOMMAND)       printf("RCmd,");
if (qual & IEQUALIFIER_NUMERICPAD)     printf("NumPad,");
if (qual & IEQUALIFIER_REPEAT)         printf("Rpt,");
if (qual & IEQUALIFIER_INTERRUPT)      printf("Intrpt,");
if (qual & IEQUALIFIER_MULTIBROADCAST) printf("Multi Broadcast,");
if (qual & IEQUALIFIER_MIDBUTTON)      printf("MidBtn,");
if (qual & IEQUALIFIER_RBUTTON)        printf("RBtn,");
if (qual & IEQUALIFIER_LEFTBUTTON)     printf("LBtn,");
if (qual & IEQUALIFIER_RELATIVEMOUSE)  printf("RelMouse,");
}

/* doKeys() - Show what keys were pressed. */
BOOL doKeys(struct IntuiMessage *msg, struct InputEvent *ievent,
            UBYTE **buffer, ULONG *bufsize)
{
USHORT char_pos;
USHORT numchars;
BOOL   ret_code = TRUE;
UBYTE  realc, c;

/* deadKeyConvert() returns -1 if there was not enough space in the buffer to
** convert the string. Here, the routine increases the size of the buffer on the
** fly...Set the return code to FALSE on failure.
*/
numchars = deadKeyConvert(msg, *buffer, *bufsize - 1, NULL, ievent);
while ((numchars == -1) && (*buffer != NULL)) {
    /* conversion failed, buffer too small. try to double the size of the buffer. */
    FreeMem(*buffer, *bufsize);
    *bufsize = *bufsize << 1;
    printf("Increasing buffer size to %d\n", *bufsize);

    if (NULL == (*buffer = AllocMem(*bufsize, MEMF_CLEAR)))  ret_code = FALSE;
    else  numchars = deadKeyConvert(msg, *buffer, *bufsize - 1, NULL, ievent);
    }

/* numchars contains the number of characters placed within the buffer.  Key up events and   */
/* key sequences that do not generate any data for the program (like deadkeys) will return   */
/* zero.  Special keys (like HELP, the cursor keys, FKeys, etc.) return multiple characters  */
/* that have to then be parsed by the application.  Allocation failed above if buffer is NULL*/
if (*buffer != NULL) {
    /* if high bit set, then this is a key up otherwise this is a key down */
    if (msg->Code & 0x80)
          printf("Key Up:   ");
    else
          printf("Key Down: ");
```

```
    print_qualifiers(msg->Qualifier);
    printf(" rawkey #%d maps to %d ASCII character(s)\n", 0x7F & msg->Code, numchars);
    for (char_pos = 0; char_pos < numchars; char_pos++) {
        realc = c = (*buffer)[char_pos];
        if ((c <= 0x1F)||((c >= 0x80)&&(c < 0xa0)))
            c = 0x7f;
        printf("  %3d ($%02x) = %c\n", realc, realc, c);
        }
    }
return(ret_code);
}

/* process_window() - simple event loop.  Note that the message is not replied
** to until the end of the loop so that it may be used in the doKeys() call.
*/
VOID process_window(struct Window *win, struct InputEvent *ievent,
    UBYTE **buffer, ULONG *bufsize)
{
struct IntuiMessage *msg;
BOOL done;

done = FALSE;
while (done == FALSE) {
    Wait((1L<<win->UserPort->mp_SigBit));
    while ((done == FALSE) && (msg = (struct IntuiMessage *)GetMsg(win->UserPort))) {
        switch (msg->Class) {     /* handle our events */
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            case IDCMP_RAWKEY:
                if (FALSE == doKeys(msg,ievent,buffer,bufsize))
                    done = TRUE;
                break;
            }
        ReplyMsg((struct Message *)msg);
        }
    }
}
```

KEYBOARD CONTROL OF THE POINTER

All Intuition mouse activities can be emulated using the keyboard, by combining the Amiga command keys with other keystrokes.

The pointer can be moved by holding down either Amiga key along with one of the four cursor keys. The mouse pointer accelerates the longer these keys are held down. Additionally, holding down either Shift key will make the pointer jump in larger increments. The pointer position may also be adjusted in very fine increments through this technique. By holding down either Amiga key and briefly pressing one of the cursor keys, the pointer may be moved one pixel in any direction.

Press the left Alt key and either one of the Amiga keys simultaneously emulates the left button of the mouse. Similarly, pressing the right Alt key and either one of the Amiga keys simultaneously emulates the right button of the mouse. These key combinations permit users to make gadget selections and perform menu operations using the keyboard alone.

INTUITION KEYBOARD SHORTCUTS

If Intuition sees a command key sequence that means nothing to it, the key sequence is sent to the active window as usual. See the ''Intuition Input and Output Methods'' section for how this works. This section and the next section describe what Intuition does when it recognizes certain special command key sequences.

It is recommended that programs abide by certain command key standards to provide a consistent interface for Amiga users. The *Amiga User Interface Style Guide* contains a complete list of the recommended standards.

There are a number of special keyboard shortcuts supported by Intuition. These involve holding down the left Amiga key and simultaneously pressing a another key. These functions allow the user to do such things as move the Workbench screen to the front using the keyboard.

Table 10-2: Intuition Keyboard Shortcuts

Keyboard Shortcut	Function Performed
left Amiga M	Move frontmost screen to back.
left Amiga N	Move Workbench screen to front.
left Amiga B	System requester cancel, or select the rightmost button in the system requester.
left Amiga V	System requester OK, or select the leftmost button in the system requester.
left Amiga + mouse select button	Screen drag from any point. By holding down the left Amiga key, the user may drag the screen with the mouse from any part of the screen or window on the screen.

About System Keyboard Shortcuts. Many of these keyboard commands may be remapped through the IControl Preferences editor. Do not rely on the values reported here.

Intuition consumes these command key sequences for its own use. That is, it always detects these events and removes them from the input stream. The application will not see the events.

MENU SHORTCUTS

Menu items and sub-items may be paired with command key sequences to associate certain characters with specific menu item selections. This gives the user a shortcut method to select frequently used menu operations, such as Undo, Cut, and Paste. Whenever the user presses the right Amiga key with an alphanumeric key, the menu strip of the active window is scanned to see if there are any command key sequences in the list that match the sequence entered by the user. If there is a match, Intuition translates the key combination into the appropriate menu item number and transmits the menu number to the application program.

Menu Shortcuts Look Like the Real Thing. To the application it looks as if the user had selected a given menu item with the mouse. The program will receive a menu event, not a key event. For more information on menu item selection, see the ''Intuition Menus'' chapter.

AMIGA QUALIFIERS

The Amiga keyboard has several special qualifiers which are listed in the next table. Most of these qualifiers are associated with special keys on the keyboard such as the Shift or Ctrl key. These keys are used to modify the meaning of other keys. Other qualifiers are associated with mouse button status. For a complete list of all the qualifiers, see the include file *<devices/inputevent.h>*.

The **Qualifier** field of each **IntuiMessage** contains the status of all the qualifiers. An individual application should never attempt to track the state of any of the qualifier keys or mouse buttons even though key-down and key-up information may be available. Instead use the information available in the **Qualifier** field of the **IntuiMessage** structure.

Table 10-3: Keyboard Qualifiers

Qualifier Type	Key Label	Explanation
Control	Ctrl	The IEQUALIFIER_CONTROL bit indicates that the Control key is depressed.
Amiga	Fancy A	There are two Amiga keys, one on each side of the space bar. The left Amiga key is recognized by the **Qualifier** bit IEQUALIFIER_LCOMMAND, and the right Amiga key by IEQUALIFIER_RCOMMAND.
Alternate	Alt	There are two separate Alt keys, one on each side of the space bar, next to the Amiga keys. These can be treated separately, if desired. The left Alt key sets the IEQUALIFIER_LALT bit and the right Alt key sets the IEQUALIFIER_RALT bit.
Shift	Up Arrow	There are two separate Shift keys, one above each Alt key. These can be treated distinctly, if desired. The left Shift key sets the IEQUALIFIER_LSHIFT bit and the right Shift key sets the IEQUALIFIER_RSHIFT bit.
Caps Lock	Caps Lock	The IEQUALIFIER_CAPSLOCK bit is set as long as the Caps Lock light is illuminated.
Numeric Pad		The IEQUALIFIER_NUMERICPAD bit is set for keys on the numeric keypad.
Repeat		Repeat key events are sent with the IEQUALIFIER_REPEAT bit set.
Mouse Buttons		If mouse buttons are down when the event occurs, one or more of the three bits IEQUALIFIER_LEFTBUTTON, IEQUALIFIER_MIDBUTTON or IEQUALIFIER_RBUTTON will be set.

Function Reference

The following are brief descriptions of the Intuition functions that relate to the use of the mouse and keyboard under Intuition. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 10-4: Functions for Intuition Mouse and Keyboard

Function	Description
DoubleClick()	Test two time values for double click status.
SetPointer()	Change the Intuition pointer imagery for an open window.
ClearPointer()	Restore the default Intuition pointer imagery.
SetMouseQueue()	Change the mouse queue for an open window.
ReportMouse()	A function C programmers should not use.

Chapter 11
INTUITION SPECIAL FUNCTIONS

There are several Intuition topics which, while not large enough to fill chapters of their own, nontheless deserve to be discussed. The subjects covered here include locking **IntuitionBase**, the Intuition memory functions **AllocRemember()** and **FreeRemember()**, using sprites with Intuition, and Intuition's special internal functions.

Locking IntuitionBase

It is sometimes necessary to examine the **IntuitionBase** structure. Items such as the address of the active screen and window, current mouse coordinates and more can be found there. It is never a good idea to simply read these fields, as they are prone to sudden change. The **IntuitionBase** structure must always be locked before looking at its fields.

It is necessary to inform Intuition that an application is about to examine **IntuitionBase** so that Intuition will not change any variables and **IntuitionBase** will remain static during the access. The call **LockIBase()** will lock the state of **IntuitionBase** so that it may be examined. During the time that the application has **IntuitionBase** locked, all Intuition input processing is frozen. Make every effort to examine **IntuitionBase** and release the lock as quickly as possible. The values in **IntuitionBase** are read-only. Applications should never write values to **IntuitionBase**.

```
ULONG LockIBase( unsigned long dontknow );
```

LockIBase() is passed a ULONG (**dontknow** in the prototype above) indicating the Intuition lock desired. For all foreseeable uses of the call this value should be 0. **LockIBase()** returns a ULONG, that must be passed to **UnlockIBase()** later to allow **IntuitionBase** to change once again.

Every call to **LockIBase()** must be matched by a subsequent call to **UnlockIBase()**:

```
void UnlockIBase( unsigned long ibLock );
```

Set the **ibLock** argument to the value returned by the previous call to **LockIBase()**.

About LockIBase(). This function should not be called while holding any other system locks such as **Layer** and **LayerInfo** locks. Between calls to **LockIBase()** and **UnlockIBase()**, you may not call any Intuition or other high-level system functions so it is best to copy the information you need and release the lock as quickly as possible.

About IntuitionBase. Never, ever, modify any of the fields in **IntuitionBase** directly. Also, there are fields in **IntuitionBase** that are considered system private that should not be accessed, even for reading. (Refer to the *<intuition/intuitionbase.h>* include file.) Application programs cannot depend on (and should not use) the contents of these fields; their usage is subject to change in future revisions of Intuition.

Easy Memory Allocation and Deallocation

Intuition has a pair of routines that enable applications to make multiple memory allocations which are easily deallocated with a single call. The Intuition routines for memory management are **AllocRemember()** and **FreeRemember()**. These routines rely upon the **Remember** structure to track allocations.

INTUITION HELPS YOU REMEMBER

The **AllocRemember()** routine calls the Exec **AllocMem()** function to perform the memory allocation. (Of course, the application may directly call Exec memory functions, see the chapter "Exec Memory Allocation" for details.)

AllocRemember() performs two allocations each time it is called. The first allocation is the actual memory requested by the application. This memory is of the size and type specified in the call and is independent of the second block of memory. The second allocation is memory for a **Remember** structure which is used to save the specifics of the allocation in a linked list. When **FreeRemember()** is called it uses the information in this linked list to free all previous memory allocations at once. This is convenient since normally you would have to free each memory block one at a time which requires knowing the size and base address of each one.

The **AllocRemember()** call takes three arguments:

```
APTR AllocRemember( struct Remember **rememberKey, unsigned long size, unsigned long flags );
```

The **rememberKey** is the address of a pointer to a **Remember** structure. Note that this is a double indirection, not just a simple pointer. The **size** is the size, in bytes, of the requested allocation. The **flags** argument is the specification for the memory allocation. These are the same as the specifications for the Exec **AllocMem()** function described in the chapter on "Exec Memory Allocation".

If **AllocRemember()** succeeds, it returns the address of the allocated memory block. It returns a NULL if the allocation fails.

The **FreeRemember()** function gives the option of freeing memory in either of two ways. The first (and most useful) option is to free both the link nodes that **AllocRemember()** created and the memory blocks to which they correspond. The second option is to free only the link nodes, leaving the memory blocks for further use (and later deallocation via Exec's **FreeMem()** function). But, as a general rule, the application should never free only the link nodes as this can greatly fragment memory. If the link nodes are not required, use the Exec memory allocation functions.

The **FreeRemember()** call is as follows:

```
void FreeRemember( struct Remember **rememberKey, long reallyForget );
```

Set the **rememberKey** argument to the address of a pointer to a **Remember** structure. This is the same value that was passed to previous calls to **AllocRemember()**. The **reallyForget** argument is a boolean that should be set to TRUE. If TRUE, then both the link nodes and the memory blocks are freed. If FALSE, then only the link nodes are freed. Again, applications should avoid using the FALSE value since it can lead to highly fragmented memory.

HOW TO REMEMBER

To use Intuition's memory functions, first create an anchor for the memory to be allocated by declaring a variable that is a pointer to a **Remember** structure and initializing that pointer to NULL. This variable is called the *remember key*.

```
struct Remember *rememberKey = NULL;
```

Call **AllocRemember()** with the address of the remember key, along with the memory requirements for the specific allocation. Multiple allocations may be made before a call to **FreeRemember()**.

```
memBlockA = AllocRemember(&rememberKey, SIZE_A, MEMF_CLEAR | MEMF_PUBLIC);
if (memBlockA == NULL)
    {
    /* error: allocation failed */
    printf("Memory allocation failed.\0);
    }
else
    {
    /* use the memory here */
    printf("Memory allocation succeeded.\0);
    }
```

AllocRemember() actually performs two memory allocations per call, one for the memory requested and the other for a **Remember** structure. The **Remember** structure is filled in with data describing the allocation, and is linked into the list to which the *remember key* points.

To free memory that has been allocated, simply call **FreeRemember()** with the correct remember key.

```
void FreeRemember(&rememberKey, TRUE);
```

This will free all the memory blocks previously allocated with **AllocRemember()** in a single call.

THE REMEMBER STRUCTURE

The **Remember** structure is defined in *<intuition/intuition.h>* as follows:

```
struct Remember
    {
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
    };
```

Generally, the **Remember** structure is handled only by the system. Here are its fields:

NextRemember
The link to the next **Remember** structure.

RememberSize
The size of the memory tracked by this node.

Memory
A pointer to the memory tracked by this node.

AN EXAMPLE OF REMEMBERING

```
;/* remembertest.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 remembertest.c
Blink FROM LIB:c.o,remembertest.o TO remembertest LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** RememberTest - Illustrates the use of AllocRemember() and FreeRemember().
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <intuition/intuition.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#include <stdlib.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* our function prototypes */
VOID methodOne(VOID);
VOID methodTwo(VOID);

struct IntuitionBase *IntuitionBase;

/* random sizes to demonstrate the Remember functions. */
#define SIZE_A 100L
#define SIZE_B 200L


/*
** main() - Initialize everything.
*/
VOID main(int argc, char **argv)
{
LONG exitVal = RETURN_OK;

IntuitionBase = OpenLibrary("intuition.library", 33L);
if (IntuitionBase == NULL)
    exitVal = RETURN_FAIL;
else
    {
    methodOne();
    methodTwo();

    CloseLibrary(IntuitionBase);
    }
exit(exitVal);
}
```

```
/*
** MethodOne
** Illustrates using AllocRemember() to allocate all memory and
** FreeRemember() to free it all.
*/
VOID methodOne(VOID)
{
APTR memBlockA = NULL, memBlockB = NULL;
struct Remember *rememberKey = NULL;

memBlockA = AllocRemember(&rememberKey, SIZE_A, MEMF_CLEAR | MEMF_PUBLIC);
if (memBlockA)
    {
    /*  The memBlockA allocation succeeded; try for memBlockB.  */
    memBlockB = AllocRemember(&rememberKey, SIZE_B, MEMF_CLEAR | MEMF_PUBLIC);
    if (memBlockB)
        {
        /*  Both memory allocations succeeded.
        **  The program may now use this memory.
        */
        }
    }

/* It is not necessary to keep track of the status of each allocation.
** Intuition has kept track of all successful allocations by updating its
** linked list of Remember nodes.  The following call to FreeRemember() will
** deallocate any and all of the memory that was successfully allocated.
** The memory blocks as well as the link nodes will be deallocated because
** the "ReallyForget" parameter is TRUE.
**
** It is possible to have reached the call to FreeRemember()
** in one of three states.  Here they are, along with their results.
**
** 1. Both memory allocations failed.
**       RememberKey is still NULL.  FreeRemember() will do nothing.
** 2. The memBlockA allocation succeeded but the memBlockB allocation failed.
**       FreeRemember() will free the memory block pointed to by memBlockA.
** 3. Both memory allocations were successful.
**       FreeRemember() will free the memory blocks pointed to by
**       memBlockA and memBlockB.
*/
FreeRemember(&rememberKey, TRUE);
}

/*
** MethodTwo
** Illustrates using AllocRemember() to allocate all memory,
** FreeRemember() to free the link nodes, and FreeMem() to
** free the actual memory blocks.
*/
VOID methodTwo(VOID)
{
APTR memBlockA = NULL, memBlockB = NULL;
struct Remember *rememberKey = NULL;

memBlockA = AllocRemember(&rememberKey, SIZE_A, MEMF_CLEAR | MEMF_PUBLIC);
if (memBlockA)
    {
    /*  The memBlockA allocation succeeded; try for memBlockB.  */
    memBlockB = AllocRemember(&rememberKey, SIZE_B, MEMF_CLEAR | MEMF_PUBLIC);
    if (memBlockB)
        {
        /* Both memory allocations succeeded.
        ** For the purpose of illustration, FreeRemember() is called at
        ** this point, but only to free the link nodes.  The memory pointed
        ** to by memBlockA and memBlockB is retained.
        */
        FreeRemember(&rememberKey, FALSE);

        /* Individually free the two memory blocks. The Exec FreeMem()
        ** call must be used, as the link nodes are no longer available.
        */
        FreeMem((VOID *)memBlockA, SIZE_A);
        FreeMem((VOID *)memBlockB, SIZE_B);
        }
    }
```

```
/* It is possible to have reached the call to FreeRemember()
** in one of three states.  Here they are, along with their results.
**
** 1. Both memory allocations failed.
**    RememberKey is still NULL.  FreeRemember() will do nothing.
** 2. The memBlockA allocation succeeded but the memBlockB allocation failed.
**    FreeRemember() will free the memory block pointed to by memBlockA.
** 3. Both memory allocations were successful.
**    If this is the case, the program has already freed the link nodes
**    with FreeRemember() and the memory blocks with FreeMem().
**    When FreeRemember() freed the link nodes, it reset RememberKey
**    to NULL.  This (second) call to FreeRemember() will do nothing.
*/
FreeRemember(&rememberKey, TRUE);
}
```

Current Time Values

The function **CurrentTime()** gets the current time values. To use this function, first declare the variables **Seconds** and **Micros**. Then, when the application call the function, the current time is copied into the argument pointers.

```
void CurrentTime( ULONG *seconds, ULONG *micros );
```

See the DOS library Autodocs in the *AmigaDOS Manual* (Bantam Books) for more information on functions dealing with the date and time. The DOS library includes such functions as **DateToStr()**, **StrToDate()**, **SetFileDate()** and **CompareDates()**.

Using Sprites in Intuition Windows and Screens

Sprite functionality has limitations under Intuition. The hardware and graphics library sprite systems manage sprites independently of the Intuition display. In particular:

- Sprites cannot be attached to any particular screen. Instead, they always appear in front of every screen.

- When a screen is moved, the sprites do not automatically move with it. The sprites move to their correct locations only when the appropriate function is called (either **DrawGList()** or **MoveSprite()**).

Hardware sprites are of limited use under the Intuition paradigm. They travel out of windows and out of screens, unlike all other Intuition mechanisms (except the Intuition pointer, which is meant to be global).

Remember that sprite data must be in Chip memory to be accessible to the custom chips. This may be done with a compiler specific feature, such as the *__chip* keyword of SAS/C. Otherwise, Chip memory can be allocated with the Exec **AllocMem()** function or the Intuition **AllocRemember()** function, setting the memory requirement flag to MEMF_CHIP. The sprite data may then be copied to Chip memory using a function like **CopyMem()** in the Exec library. See the chapter ''Graphics Sprites, Bobs and Animation'' for more information.

Intuition and Preferences

The **SetPrefs()** function is used to configure Intuition's internal data states according to a given Preferences structure. This call relies on the Preferences system used in V34 and earlier versions of the OS. The old system has been largely superceded in Release 2. See the "Preferences" chapter for details. This routine is called only by:

- The Preferences program itself after the user changes Preferences settings (under V34 and earlier).
- AmigaDOS when the system is being booted up. AmigaDOS opens the *devs:system-configuration* file and passes the information found there to the **SetPrefs()** routine. This way, the user can create an environment and have that environment restored every time the system is booted.

The function takes three arguments:

```
struct Preferences *SetPrefs(struct Preferences *prefbuf, long size, long realThing)
```

The **prefbuf** argument is a pointer to a Preferences structure that will be used for Intuition's internal settings. The **size** is the number of bytes contained in your Preferences structure. Typically, you will use **sizeof(struct Preferences)** for this argument. The **realThing** argument is a boolean TRUE or FALSE designating whether or not this is an intermediate or final version of the Preferences. The difference is that final changes to Intuition's internal Preferences settings cause a global broadcast of NEWPREFS events to every application that is listening for this event. Intermediate changes may be used, for instance, to update the screen colors while the user is playing with the color gadgets.

> *About SetPrefs().* The intended use for the **SetPrefs()** call is entirely to serve the user. You should never use this routine to make your programming or design job easier at the cost of yanking the rug out from beneath the user.

Refer to the chapter "Preferences" for information about the **Preferences** structure and the new Preferences procedure calls used in Release 2.

Function Reference

The following are brief descriptions of the Intuition functions discussed in this chapter. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 11-1: Other Functions for Intuition

Function	Description
AllocRemember()	Allocate memory and track the allocation.
FreeRemember()	Free memory allocated with **AllocRemember()**.
LockIBase()	Lock IntuitionBase for reading.
UnlockIBase()	Unlock IntuitionBase when done reading.
CurrentTime()	Get the system time in seconds and micro-seconds.
SetPrefs()	An Intuition internal function you should try to avoid.

Chapter 12
BOOPSI—OBJECT ORIENTED INTUITION

Boopsi is an acronym for *Basic Object Oriented Programming System for Intuition.* Using the Object Oriented Programming (OOP) model, Boopsi represents certain Intuition entities, like **Gadgets** and **Images**, as objects.

There are many advantages to using Boopsi:

- Boopsi makes Intuition customizable and extensible. Boopsi programmers can create new types of Boopsi objects to suit the needs of their applications. These new types of objects are part of Intuition and can be made public so other applications can use them. Because applications can share the new types, application writers don't have to waste their time duplicating each other's efforts writing the same objects.

- New types of Boopsi objects can build on old types of Boopsi objects, inheriting the old object's behavior. The result is that Boopsi programmers don't have to waste their time building new objects from scratch, they simply add to the existing object.

- OOP and Boopsi apply the concept of interchangeable parts to Intuition programming. A Boopsi programmer can combine different Boopsi objects (like gadgets and images) to create an entire Graphical User Interface (GUI). The Boopsi programmer doesn't have take the time to understand or implement the inner workings of these objects. The Boopsi programmer only needs to know how to interact with Boopsi objects and how to make them interact with each other.

- Boopsi objects have a consistent, command-driven interface. To the Boopsi programmer, there is no difference between displaying a text, border, or bitmap-based Boopsi image, even though they are rendered quite differently. Each image object accepts a single command to tell it to render itself.

Before reading this chapter, you should already be familiar with several Amiga concepts. Boopsi is built on top of Intuition and uses many of its structures. These include Intuition gadgets, images, and windows. Boopsi also uses the tag concept to pass parameters. The "Utility Library" chapter of this manual discusses tags. The "Utility Library" chapter also discusses callback **Hooks**, which are important to the later sections of this chapter.

OOP Overview

Understanding Boopsi requires an understanding of several of the concepts behind Object Oriented Programming. This section is a general overview of these concepts as they pertain to Boopsi. Because Boopsi is in part based on the concepts present in the OOP language Smalltalk, a reference book on Smalltalk may provide a deeper understanding of Boopsi in general. Timothy Budd's book entitled *A Little Smalltalk* (Addison-Wesley Publishing ISBN 0-201-10698-1) is a good start.

In the Boopsi version of the Object Oriented Programming model, everything is an *Object*. For example, a proportional gadget named **myprop** is an object. Certain objects have similar characteristics and can be classified into groups called *classes*. As objects, Rover the dog, Bob the cat, and Sam the bird are all distinct objects but they all have something in common, they can all be classified as animals. As objects, **myprop** the proportional gadget, **mystring** the string gadget, and **mybutton** the button gadget all have something in common, they can all be classified as gadgets. A specific object is an *instance* of a particular class (''Rover'' is an instance of class ''animal'', ''myslidergadget'' is an instance of class ''gadget'').

Notice that, although Rover, Bob, and Sam can all be classified as animals, each belongs to a subgroup of the animal class. ''Rover'' is an instance of class ''dog'', ''Bob'' is an instance of class ''cat'', and ''Sam'' is an instance of class ''bird''. Because each of these animal types share common characteristics, each type makes up its own class. Because dog, cat, and bird are subclassifications of the animal class, they are known as *subclasses* of the animal class. Conversely, the animal class is the *superclass* of the dog, cat, and bird classes.

Following the branches upward from class to superclass will bring you to a universal root category from which all objects are derived. The OOP language Smalltalk calls this class ''Object''.

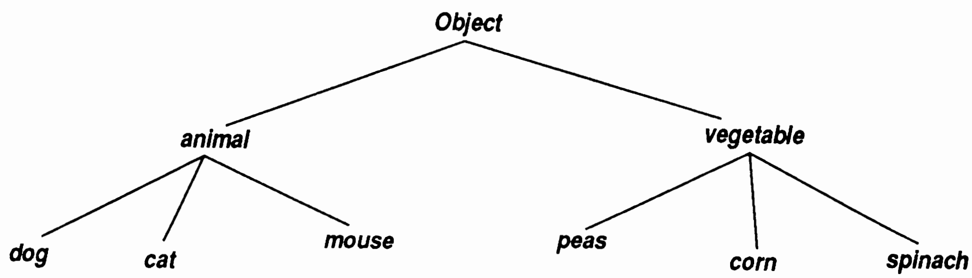


Figure 12-1: Object Diagram

Like Smalltalk, Boopsi also has a universal root catagory, *rootclass*. Currently, Intuition defines three immediate subclasses of rootclass. The first, *gadgetclass*, is the class of Boopsi gadgets. The second class, *imageclass*, makes up the class of Boopsi images.

Unlike gadgetclass and imageclass, the remaining subclass, *icclass*, does not correspond to an existing Intuition entity, it is a concept new to Intuition. Icclass, or *interconnection class*, allows one Boopsi object to notify another Boopsi object when a specific event occurs. For example, consider a Boopsi proportional gadget and a Boopsi image object that displays an integer value. An application can connect these two objects so that the prop gadget tells the image object the prop gadget's current value, which the image object displays. Every time the user slides the prop gadget, the prop gadget notifies the image of the change and the image updates its display to reflect the prop gadget's current integer value. Because these objects are talking to each other rather than the application, the updates happen automatically. The application doesn't have to talk to the two objects, it only has to connect them.

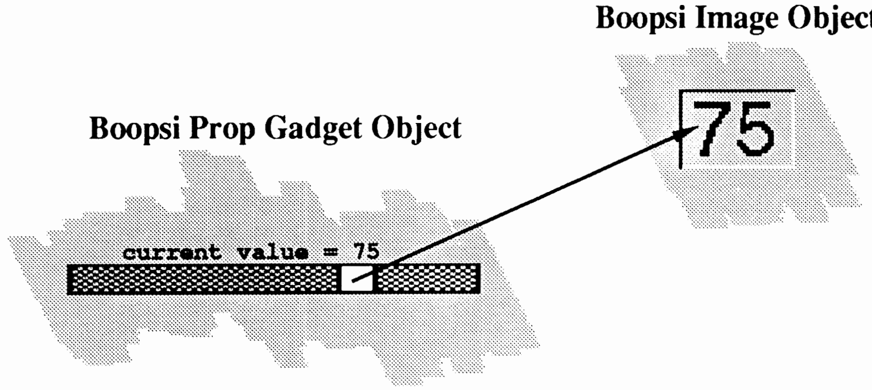


Figure 12-2: Simple Boopsi Diagram

An object's characteristics and behavior are determined by its class. Each class can define a set of *attributes* and a set of *methods* that apply to all objects of that class. An attribute is a variable characteristic of an object. For example, an attribute for the animal class could be the number of legs an animal object has. An example of a Boopsi attribute is the X coordinate of a Boopsi image object. The data that makes up the values of an object's attributes is collectively known as the *instance data* for that object.

The behavior of an object depends upon the set of *methods* associated to it by its class. A method is basically a function that applies to objects of that class. An example of a Boopsi method is the imageclass method IM_DRAW. This method tells a Boopsi image to draw itself. All Boopsi actions are carried out via methods.

From the Object Diagram, two of the methods of the "animal" class could be "eat" and "sleep". One of the methods of the "dog" class could be "bark". Notice that instances of the "dog" class can do more than just bark, they can also eat and sleep. This is because a subclass *inherits* methods from its superclasses. If there were a subclass of dog called "attack dog", all instances of that class would be able to bark, eat, and sleep, as well as "attack". Due to inheritance, a subclass has all of the methods and all of the attributes of its superclass. For example, the IA_Height attribute is defined by imageclass. All instances of the subclasses of imageclass have their own IA_Height attribute, even though the subclasses do not explicitly define IA_Height. In turn, all instances of subclasses of the imageclass subclasses also inherit the IA_Height attribute. All classes on levels below a class will inherit its methods and attributes.

When an application or a Boopsi object wants another Boopsi object to perform a method, it passes it a command in the form of a Boopsi *message*. A Boopsi message tells an object which method to perform. The message may also contain some parameters that the method requires.

> *Watch Out!* The term "message" used in object oriented terminology can be little confusing to the Amiga programmer because the Boopsi message has nothing to do with an Exec message.

Boopsi classes can be either *public* or *private*. Public classes have ASCII names associated with them and are accessible to all applications. Private classes have no ASCII name and normally can only be accessed by the application that created the private class.

USING BOOPSI

There are several levels on which a programmer can use Boopsi. The most elementary level is to use Intuition functions to create and manipulate Boopsi objects that are instances of existing, public classes.

At present there is a hierarchy of 14 public classes built into Intuition:

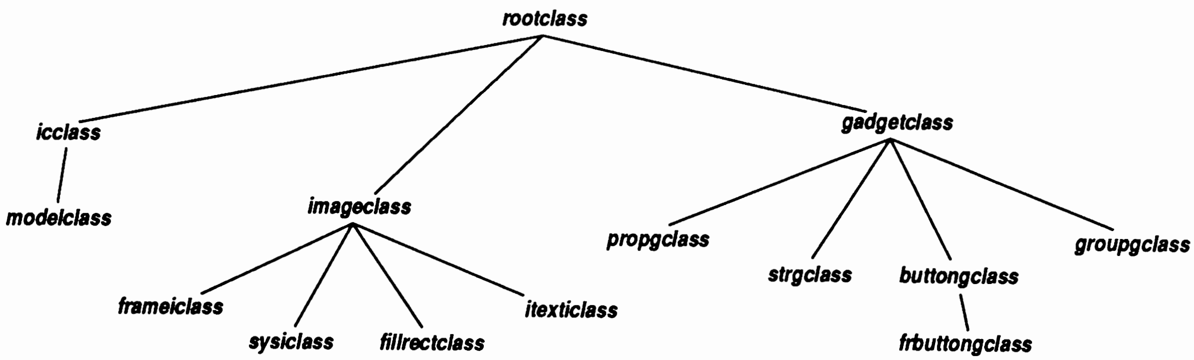


Figure 12-3: Class Diagram

Boopsi and Tags

Boopsi uses tag lists to pass and manipulate its attributes. To Boopsi, each **TagItem** (defined in <*utility/tagitem.h*>) in a tag list is an attribute/value pair. The **TagItem.ti_Tag** field contains an ID for the attribute and the **ti_Data** field holds the attribute's value.

For example, the string gadget class defines an attribute called STRINGA_LongVal, which is the current integer value of the gadget. Certain gadgetclass objects have an attribute called GA_Image. Its value is not an integer, it is a pointer to an image.

Note that these tag lists can also contain utility.library Global System control tags (like TAG_SKIP and TAG_DONE), which Boopsi uses in processing its tag lists. Any application that ends up processing these lists should do so using the tag manipulation functions from utility.library. For more information on tags and utility.library, see the "Utility Library" chapter of this manual.

Creating an Object

The Intuition function **NewObjectA()** creates a Boopsi object:

```
mynewobject = APTR NewObjectA(Class *privclass, UBYTE *pubclass, struct TagItem *myattrs)
```

The pointer that **NewObjectA()** returns is a pointer to a Boopsi object. In general, Boopsi objects are "black boxes". This means the inner workings of Boopsi objects are not visible to the application programmer, so the programmer does not know what goes on inside it. This really means the inner workings of these objects are none of your business. Unless otherwise documented, only use an object pointer as a handle to the object.

To create an object, **NewObjectA()** needs to know what class the new object is an instance of. To create a public class object, pass a NULL pointer in **privclass** and an ASCII string in **pubclass** naming the object's public class. The **privclass** pointer is used to create a private class object, which is covered in the "Creating a Boopsi Class" section later in this chapter.

The **myattrs** tag list is a list of tag/value pairs, each of which contains an initial value for some object attribute. Most objects have a set of attributes associated with them, so each attribute has a tag name. For Boopsi gadgets and images, the attributes include some of the values from the old **Gadget** and **Image** structures (position, size, etc.).

Most applications use the stack-based version of **NewObjectA()**, **NewObject()**, to create objects. This allows an application to build the tag list of object attributes on the stack rather than having to allocate and initialize a tag list. A code sample from a program that creates a Boopsi string gadget might look like this:

```
mystringgadget = (struct Gadget *)NewObject(NULL, "strgclass", GA_ID,          1L,
                                                               GA_Left,        0L,
                                                               GA_Top,         0L,
                                                               STRINGA_LongVal, 100L,
                                                               TAG_END);
```

If **NewObject()** is successful, it returns a pointer to a new Boopsi gadget object. Otherwise, it returns NULL. The class "strgclass" is one of the public classes built into Release 2. It is a class of string gadgets.

If you look at the diagram of the public classes built into Intuition, you'll see that strgclass is a subclass of gadgetclass. In the example above, the attribute tag IDs that start with "GA_" are defined by gadgetclass and not by strgclass. This is because strgclass inherits these attributes from its superclass, gadgetclass. The other attribute, STRINGA_LongVal, is defined by strgclass. It does two things. First, it tells the object that it is a special type of string gadget which only handles an integer value rather than a generic ASCII string. Second, it passes the object its initial integer value.

Disposing of an Object

When an application is done with an object it has to dispose of the object. To dispose of an object, use the Intuition function **DisposeObject()**:

```
VOID DisposeObject(APTR boopsiobject);
```

where **boopsiobject** is a pointer to the Boopsi object to be disposed. Note that some classes allow applications to connect child objects to a parent object so that when the application deletes the parent object, it automatically disposes of all of its children. Be careful not to dispose of an object that has already been disposed.

Setting an Existing Object's Attributes

An object's attributes are not necessarily static. An application can ask an object to set certain object attributes using the **SetAttrs()** function:

```
ULONG SetAttrs(APTR myobject, Tag1, Value1, ...);
```

Because Boopsi gadgets require some extra information about their display, they use a special version of this function, **SetGadgetAttrs()**:

```
ULONG SetGadgetAttrs(struct Gadget *myobject, struct Window *w, struct Requester *r,
                     Tag1, Value1, ...);
```

Here **myobject** is a pointer to the Boopsi object, **w** points to the gadget's window, **r** points to the gadget's requester, and the tag/value pairs are the attributes and their new values. The return value of **SetAttrs()** and **SetGadgetAttrs()** is class specific. In general, if the attribute change causes a visual change to some object, the **SetAttrs()/SetGadgetAttrs()** function should return a non-zero value, otherwise, these functions should return zero (see the *Boopsi Class Reference* in "Appendix B" of this manual for information on the return values for specific classes). The following is an example of how to set the current integer value and gadget ID of the gadget created in the **NewObject()** call above:

```
SetGadgetAttrs(mystringgadget, mywindow, NULL, STRINGA_LongVal,   75L,
                                               GA_ID,             2L,
                                               TAG_END));
```

This changes two of **mystringgadget**'s attributes. It changes the gadget's current integer value to 75 and it changes the gadget's ID number to 2.

Note that it is not OK to call **SetGadgetAttrs()** on a Boopsi object that isn't a gadget, nor is it OK to call **SetAttrs()** on a Boopsi gadget.

Not all object attributes can be set with **SetGadgetAttrs()/SetAttrs()**. Some classes are set up so that applications cannot change certain attributes. For example, the imagery for the knob of a proportional gadget cannot be altered after the object has been created. Whether or not a specific attribute is "settable" is class dependent. For more information about the attributes of specific classes, see the *Boopsi Class Reference* in the Appendix B of this manual.

Getting an Object's Attributes

The Intuition function **GetAttr()** asks an object what the value of a specific attribute is:

```
ULONG GetAttr(ULONG attrID, APTR myobject, ULONG *mydata);
```

where **attrID** is the attribute's ID number, **myobject** is the object to get the attribute from, and **mydata** points to a data area that will hold the attribute value. This function returns a 0L if the object doesn't recognize the attribute, otherwise it returns some non-zero value, the meaning of which depends on the class. In most cases, **GetAttr()** returns a 1 when it is successful.

Not all object attributes are obtainable using the **GetAttr()** function. Some classes are set up so that applications cannot query the state of certain attributes. For example, using the GA_Image attribute, an application can give a Boopsi prop gadget (propgclass) an **Image** structure which the gadget uses as the imagery for its knob. This attribute is not "gettable" as there is no need for an application to have to ask the gadget for the structure that the application passed it in the first place. Whether or not a specific attribute is "gettable" is class dependent. For more information about the attributes of specific classes, see the *Boopsi Class Reference* in the Appendix B of this manual.

What About the Boopsi Messages and Methods?

According to the "OOP Overview" section, for an object to perform a method, something has to pass it a Boopsi message. The previous section discussed using Intuition functions to ask an object to do things like set and get attributes. The functions in the previous section seem to completely ignore all that material about methods and messages. What happened to the methods and messages?

Nothing—these functions don't ignore the OOP constructs, they just shield the programmer from them. Each of these functions corresponds to a Boopsi method:

NewObject()	OM_NEW
DisposeObject()	OM_DISPOSE
SetAttrs()/SetGadgetAttrs()	OM_SET
GetAttr()	OM_GET

These methods are defined on the rootclass level, so all Boopsi classes inherit them. The Intuition functions that correspond to these methods take care of constructing and sending a Boopsi message with the appropriate method ID and parameters.

The Public Classes

Intuition contains 14 public classes, all of which are descendants of the rootclass. There are three primary classes that descend directly from rootclass: *imageclass*, *gadgetclass*, and *icclass*.

THE IMAGECLASS SUBCLASSES

Normally, an application does not create an imageclass object. Instead, it will use a subclass of imageclass. Currently, there are four subclasses: *frameiclass*, *sysiclass*, *fillrectclass*, and *itexticlass*.

frameiclass
: An embossed or recessed rectangular frame image, that renders itself using the proper **DrawInfo** pens. This class is intelligent enough to bound or center its contents.

sysiclass
: The class of system images. The class includes the images for the system and GadTools gadgets.

fillrectclass
: A class of rectangle images that have frame and patternfill support.

itexticlass
: A specialized image class used for rendering text.

For more information on these classes see the *Boopsi Class Reference* in the Appendix B of this manual. It describes all of the existing public classes, their methods, and their attributes.

The Gadgetclass Subclasses

Like imageclass, applications do not normally create objects of gadgetclass, but instead create objects of its subclasses. Currently, gadgetclass has four subclasses:

propgclass
: An easy to implement, horizontal or vertical proportional gadget.

strgclass
: A string gadget.

groupgclass
: A special gadget class that creates one composite gadget out of several others.

buttongclass
A button gadget that keeps sending button presses while the user holds it down.

buttongclass has a subclass of its own:

frbuttonclass
A buttongclass gadget that outlines its imagery with a frame.

For specific information on these classes, see the *Boopsi Class Reference* in the Appendix B of this manual.

MAKING GADGET OBJECTS TALK TO EACH OTHER

One use for a proportional gadget is to let the user change some integer value, like the red, green, and blue components of a color. This type of prop gadget is commonly accompanied by an integer string gadget, enabling the user to adjust one integer value by either typing the value into the string gadget or by scrolling the prop gadget. Because these two gadgets reflect the value of the same integer, when the user adjusts the state of one of the gadgets (and thus changing the integer value), the other gadget should automatically update to reflect the new integer value.

When the user manipulates a conventional gadget, the gadget sends messages to an IDCMP port to indicate the state change (for information on IDCMP, see the ''Intuition Input and Output Methods'' chapter of this manual). To connect the string and prop gadgets from the previous paragraph, an application would have to listen for the IDCMP messages from two different gadgets, interpret the IDCMP message's meaning, and manually update the gadgets accordingly. Essentially, the application is responsible for ''gluing'' the gadgets together. This unnecessarily complicates an application, especially when that application already has to listen for and interpret many other events.

Boopsi gadgets simplify this. By setting the appropriate attributes, an application can ask a Boopsi gadget to tell some other object when its state changes. One of the attributes defined by gadgetclass is ICA_TARGET (defined in *<intuition/icclass.h>*). The ICA_TARGET attribute points to another Boopsi object. When certain attributes in a Boopsi gadget change (like the integer value of a prop gadget), that gadget looks to see if it has an ICA_TARGET. If it does, it sends the target a message telling it to perform an OM_UPDATE method.

The OM_UPDATE method is defined by rootclass. This is basically a special type of OM_SET method that is used specifically to tell a Boopsi object that another Boopsi object's state changed. Only Boopsi objects send OM_UPDATE messages. Note that standard classes of Boopsi gadgets only send out OM_UPDATE messages as a result of the user changing the state of the gadget (scrolling the prop gadget, typing a new number into an integer gadget, etc.). These gadgets do not send out OM_UPDATE messages when they receive OM_SET or OM_UPDATE messages.

A Boopsi propgclass object has only one attribute that triggers it to send an OM_UPDATE request: PGA_Top. This attribute contains the integer value of the prop gadget. Every time the user moves a prop gadget, the PGA_Top attribute changes. If the prop gadget has an ICA_TARGET, the prop gadget will tell the target object that the PGA_Top value has changed.

A Boopsi integer string gadget (a strgclass object) also has only one attribute that triggers it to send an OM_UPDATE request: STRINGA_LongVal. This value contains the integer value of the integer string gadget. Like the prop gadget, if the integer string gadget has an ICA_TARGET, when the user changes the gadget's integer value (STRINGA_LongVal), the string gadget will tell the target object that the STRINGA_LongVal value has changed.

When a Boopsi gadget sends an OM_UPDATE message, it passes the ID of the attribute that changed plus that attribute's new value. For example, if the user typed a 25 into a Boopsi integer string gadget, that gadget would send an OM_UPDATE message to its ICA_TARGET saying in essence, "Hey, STRINGA_LongVal is 25".

If this string gadget's ICA_TARGET is a propgclass object, the propgclass object will become confused because it has no idea what a STRINGA_LongVal attribute is. The string gadget needs to *map* its STRINGA_LongVal ID to the PGA_Top ID. This is what the ICA_MAP attribute is for.

The ICA_MAP attribute is defined by gadgetclass (it is also defined for icclass—more on that later). It accepts a tag list of attribute mappings. When a gadget sends out an OM_UPDATE message, it uses this map to translate a specific attribute ID to another attribute ID, without changing the value of the attribute. Each **TagItem** in the ICA_MAP makes up a single attribute mapping. The **TagItem.ti_Tag** of the mapping is the ID of an attribute to translate. The gadget translates that attribute ID to the attribute ID in **TagItem.ti_Data**. For example, an ICA_MAP that maps a string gadget's STRINGA_LongVal attribute to a prop gadget's PGA_Top attribute looks like this:

```
struct TagItem slidertostring[] = {
    {PGA_Top, STRINGA_LongVal},
    {TAG_END, }
};
```

Note that it is OK to have an ICA_TARGET without having an ICA_MAP. In cases where a gadget and its ICA_TARGET have a set of attributes in common, it would be unnecessary to use an ICA_MAP to match a gadget's attributes, as they already match.

The following example, *Talk2boopsi.c*, creates a prop gadget and an integer string gadget which update each other without the example program having to process any messages from them.

```
;/* Talk2boopsi.c - Execute me to compile me with SAS/C 5.10b
LC -b1 -cfistq -v -y -j73 Talk2boopsi.c
Blink FROM LIB:c.o,Talk2boopsi.o TO Talk2boopsi LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;*/
/* This example creates a Boopsi prop gadget and integer string gadget, connecting them so they */
/* update each other when the user changes their value.  The example program only initializes   */
/* the gadgets and puts them on the window; it doesn't have to interact with them to make them  */
/* talk to each other.                                                                          */

#include <exec/types.h>
#include <utility/tagitem.h>
#include <intuition/intuition.h>
#include <intuition/gadgetclass.h> /* contains IDs for gadget attributes */
#include <intuition/icclass.h>     /* contains ICA_MAP, ICA_TARGET       */
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>

#ifdef LATTICE                     /* Disable SAS/C CTRL/C handling    */
int CXBRK(void)  { return(0); }
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\ $VER: Talk2boopsi 37.1";

struct Library *IntuitionBase;
struct Window *w;
struct IntuiMessage *msg;
struct Gadget *prop, *integer;

/* The attribute mapping lists */
struct TagItem prop2intmap[] =    /* This tells the prop gadget to */
{                                 /* map its PGA_Top attribute to  */
    {PGA_Top,   STRINGA_LongVal}, /* STRINGA_LongVal when it       */
    {TAG_END,}                    /* issues an update about the    */
};                                /* change to its PGA_Top value.  */
```

```
struct TagItem int2propmap[] =    /* This tells the string gadget */
{                                 /* to map its STRINGA_LongVal   */
    {STRINGA_LongVal,   PGA_Top}, /* attribute to PGA_Top when it */
    {TAG_END,}                    /* issues an update.            */
};

#define PROPGADGET_ID       1L
#define INTGADGET_ID        2L
#define PROPGADGETWIDTH     10L
#define PROPGADGETHEIGHT    80L
#define INTGADGETHEIGHT     18L
#define VISIBLE             10L
#define TOTAL               100L
#define INITIALVAL          25L
#define MINWINDOWWIDTH      80
#define MINWINDOWHEIGHT     (PROPGADGETHEIGHT + 70)
#define MAXCHARS            3L


void main(void)
{
    BOOL done = FALSE;

    if (IntuitionBase = OpenLibrary("intuition.library", 37L))
    {                              /* Open the window--notice that the window's IDCMP port     */
                                   /* does not listen for GADGETUP messages.                   */
        if (w = OpenWindowTags(NULL,
                 WA_Flags,       WFLG_DEPTHGADGET | WFLG_DRAGBAR |
                                     WFLG_CLOSEGADGET | WFLG_SIZEGADGET,
                 WA_IDCMP,       IDCMP_CLOSEWINDOW,
                 WA_MinWidth,    MINWINDOWWIDTH,
                 WA_MinHeight,   MINWINDOWHEIGHT,
                 TAG_END))
        {                                   /* Create a new propgclass object */
            if (prop = (struct Gadget *)NewObject(NULL, "propgclass",
                GA_ID,      PROPGADGET_ID,         /* These are defined by gadgetclass and        */
                GA_Top,     (w->BorderTop) + 5L,   /* correspond to similarly named fields in     */
                GA_Left,    (w->BorderLeft) + 5L,  /* the Gadget structure.                       */
                GA_Width,   PROPGADGETWIDTH,
                GA_Height,  PROPGADGETHEIGHT,

                ICA_MAP,        prop2intmap, /* The prop gadget's attribute map */

             /* The rest of this gadget's attributes are defined by propgclass. */
                PGA_Total,      TOTAL,           /* This is the integer range of the prop gadget.  */
                PGA_Top,        INITIALVAL,      /* The initial integer value of the prop gadget.  */

                PGA_Visible,    VISIBLE, /* This determines how much of the prop gadget area is    */
                                         /* covered by the prop gadget's knob, or how much of      */
                                         /* the gadget's TOTAL range is taken up by the prop       */
                                         /* gadget's knob.                                         */

                PGA_NewLook,    TRUE,    /* Use new-look prop gadget imagery */
                TAG_END))
            {                            /* create the integer string gadget.                     */
                if (integer = (struct Gadget *)NewObject(NULL, "strgclass",
                    GA_ID,       INTGADGET_ID,         /* Parameters for the Gadget structure     */
                    GA_Top,      (w->BorderTop) + 5L,
                    GA_Left,     (w->BorderLeft) + PROPGADGETWIDTH + 10L,
                    GA_Width,    MINWINDOWWIDTH -
                                   (w->BorderLeft + w->BorderRight +
                                   PROPGADGETWIDTH + 15L),
                    GA_Height,   INTGADGETHEIGHT,

                    ICA_MAP,     int2propmap,          /* The attribute map */
                    ICA_TARGET, prop,                  /* plus the target.  */

                            /* Th GA_Previous attribute is defined by gadgetclass and is used to */
                            /* wedge a new gadget into a list of gadget's linked by their         */
                            /* Gadget.NextGadget field.  When NewObject() creates this gadget,    */
                            /* it inserts the new gadget into this list behind the GA_Previous    */
                            /* gadget. This attribute is a pointer to the previous gadget         */
                            /* (struct Gadget *).  This attribute cannot be used to link new      */
                            /* gadgetsinto the gadget list of an open window or requester,        */
                    GA_Previous, prop,  /* use AddGList() instead.                                */
```

```
                STRINGA_LongVal,  INITIALVAL, /* These attributes are defined by strgclass.  */
                STRINGA_MaxChars, MAXCHARS,   /* The first contains the value of the         */
                TAG_END))                     /* integer string gadget. The second is the    */
                                              /* maximum number of characters the user is    */
                                              /* allowed to type into the gadget.            */
            {
                SetGadgetAttrs(prop, w, NULL, /* Because the integer string gadget did not   */
                    ICA_TARGET, integer,      /* exist when this example created the prop    */
                    TAG_END);                 /* gadget, it had to wait to set the           */
                                              /* ICA_Target of the prop gadget.              */

                AddGList(w, prop, -1, -1, NULL);  /* Add the gadgets to the                  */
                RefreshGList(prop, w, NULL, -1);  /* window and display them.                */

                while (done == FALSE)      /* Wait for the user to click */
                {                          /* the window close gadget.   */
                    WaitPort((struct MsgPort *)w->UserPort);
                    while (msg = (struct IntuiMessage *)
                       GetMsg((struct MsgPort *)w->UserPort))
                    {
                        if (msg->Class == IDCMP_CLOSEWINDOW)
                            done = TRUE;
                        ReplyMsg(msg);
                    }
                }
                RemoveGList(w, prop, -1);
                DisposeObject(integer);
            }
            DisposeObject(prop);
        }
        CloseWindow(w);
    }
    CloseLibrary(IntuitionBase);
  }
}
```

MAKING GADGETS TALK TO AN APPLICATION

There are two questions that the example above brings to mind. The first is, ''What happens if the user types a value into the string gadget that is beyond the bounds of the prop gadget?'' The answer is simple: very little. The prop gadget is smart enough to make sure its integer value does not go beyond the bounds of its display. In the example, the prop gadget can only have values from 0 to 90. If the user tries to type a value greater than 90, the prop gadget will set itself to its maximum of 90. Because the integer string gadget doesn't have any bounds checking built into it, the example needs to find an alternative way to check the bounds.

The other question is, ''How does *talk2boopsi.c* know the current value of the gadgets?'' That answer is simple too: it doesn't. The example doesn't ask the gadgets what their current values are (which it would do using **GetAttr()**) and the example doesn't pay attention to gadget events at the window's IDCMP port, so it isn't going to hear about them.

One easy way to hear about changes to the gadget events is to listen for a ''release verify''. Conventional Intuition gadgets can trigger a release verify IDCMP event when the user finishes manipulating the gadget. Boopsi gadgets can do this, too, while continuing to update each other.

To make *Talk2boopsi.c* do this would require only a few changes. First, the window's IDCMP port has to be set up to listen for IDCMP_GADGETUP events. Next, the example needs to set the gadget's GLFG_RELVERIFY flags. It can do this by setting the gadgetclass GA_RelVerify attribute to TRUE for both gadgets. That's enough to trigger the release verify message, so all *Talk2boopsi.c* needs to do is account for the new type of IDCMP message, IDCMP_GADGETUP. When *Talk2boopsi.c* gets a release verify message, it can use **GetAttr()** to ask the integer gadget its value. If this value is out of range, it should explicitly set the value of the integer gadget to a more suitable value using **SetGadgetAttrs()**.

Using the GLFG_RELVERIFY scheme above, an application will only hear about changes to the gadgets after the user is finished changing them. The application does not hear all of the interim updates that, for example, a prop gadget generates. This is useful if an application only needs to hear the final value and not the interim update.

It is also possible to make the IDCMP port of a Boopsi gadget's window the ICA_TARGET of the gadget. There is a special value for ICA_TARGET called ICTARGET_IDCMP (defined in *<intuition/icclass.h>*). This tells the gadget to send an IDCMP_IDCMPUPDATE class **IntuiMessage** to its window's IDCMP port. Of course, the window has to be set up to listen for IDCMP_IDCMPUPDATE **IntuiMessages**. The Boopsi gadget passes an address in the **IntuiMessage.IAddress** field. It points to an attribute tag list containing the attribute (and its new value) that triggered the IDCMP_IDCMPUPDATE message. An application can use the utility.library tag functions to access the gadget's attributes in this list. Using this scheme, an application will hear all of the interim gadget updates. If the application is using a gadget that generates a lot of interim OM_UPDATE messages (like a prop gadget), the application should be prepared to handle a lot of messages.

Using this IDCMP_IDCMPUPDATE scheme, if the gadget uses an ICA_MAP to map the attribute to a special dummy attribute ICSPECIAL_CODE (defined in *<intuition/icclass.h>*), the **IntuiMessage.Code** field will contain the value of the attribute. Because the attribute's value is a 32-bit quantity and the **IntuiMessage.Code** field is only 16 bits wide, only the least significant 16 bits of the attribute will appear in the **IntuiMessage.Code** field, so it can't hold a 32-bit quantity, like a pointer. Applications should only use the lower 16 bits of the attribute value.

THE INTERCONNECTION CLASSES

The IDCMP_IDCMPUPDATE scheme presents a problem to an application that wants to make gadgets talk to each other and talk to the application. Boopsi gadgets only have one ICA_TARGET. One Boopsi gadget can talk to either another Boopsi object or its window's IDCMP port, but not both. Using this scheme alone would force the application to update the integer value of the gadgets, which is what we are trying to avoid in the first place.

One of the standard Boopsi classes, icclass, is a class of information forwarders. An icclass object receives OM_UPDATE messages from one object and passes those messages on to its own ICA_TARGET. If it needs to map any incoming attributes, it can use its own ICA_MAP to do so.

Icclass has a subclass called *modelclass*. Using a modelclass object, an application can chain a series of these objects together to set up a "broadcast list" of icclass objects. The modelclass object is similar to the icclass object in that it has its own ICA_TARGET and ICA_MAP. It differs in that an application can use the modelclass OM_ADDMEMBER method to add icclass objects to the modelclass object's broadcast list.

The OM_ADDMEMBER method is defined by rootclass. It adds one Boopsi object to the personal list of another Boopsi object. It is up to the Boopsi object's class to determine the purpose of the objects in the list. Unlike the other methods mentioned so far in this chapter, OM_ADDMEMBER does not have an Intuition function equivalent. To pass an OM_ADDMEMBER message to an object use the *amiga.lib* function **DoMethodA()**, or its stack-based equivalent, **DoMethod()**:

```
ULONG DoMethodA(Object *myobject, Msg boopsimessage);
ULONG DoMethod(Object *myobject, ULONG methodID, ...);
```

The return value is class-dependent. The first argument to both of these functions points to the object that will receive the Boopsi message.

For **DoMethodA()**, **boopsimessage** is the actual Boopsi message. The layout of it depends on the method. Every method's message starts off with an **Msg** (from *<intuition/classusr.h>*):

```
typedef struct {
    ULONG MethodID; /* Method-specific data may follow this field */
} *Msg;
```

The message that the OM_ADDMEMBER method uses looks like this (from *<intuition/classusr.h>*):

```
struct opMember {
    ULONG    MethodID;
    Object   *opam_Object;
};
```

where **MethodID** is OM_ADDMEMBER and **opam_Object** points to the object to add to **myobject**'s list.

DoMethod() uses the stack to build a message. To use **DoMethod()**, just pass the elements of the method's message structure as arguments to **DoMethod()** in the order that they appear in the structure. For example, to ask the Boopsi object **myobject** to add the object **addobject** to its personal list:

```
DoMethod(myobject, OM_ADDMEMBER, addobject);
```

To rearrange *Talk2boopsi.c* so that it uses a modelclass object (also known as a model):

- Create the integer and prop gadget.
- Create the model.
- Create two icclass objects, one called **int2prop** and the other called **prop2int**.
- Make the model the ICA_TARGET of both the integer gadget and the prop gadget. The gadgets do not need an ICA_MAP.
- Using **DoMethod()** to call OM_ADDMEMBER, add the icclass objects to the model's personal list.
- Make the prop gadget the ICA_TARGET of **int2prop**. Make the integer gadget the ICA_TARGET of **prop2int**.
- Create an ICA_MAP map list for **int2prop** that maps STRINGA_LongVal to PGA_Top. Create an ICA_MAP map list for **prop2int** that maps PGA_Top to STRINGA_LongVal. Make the ICA_TARGET of the model ICTARGET_IDCMP.

Diagrammatically, the new *Talk2boopsi.c* should look something like this:

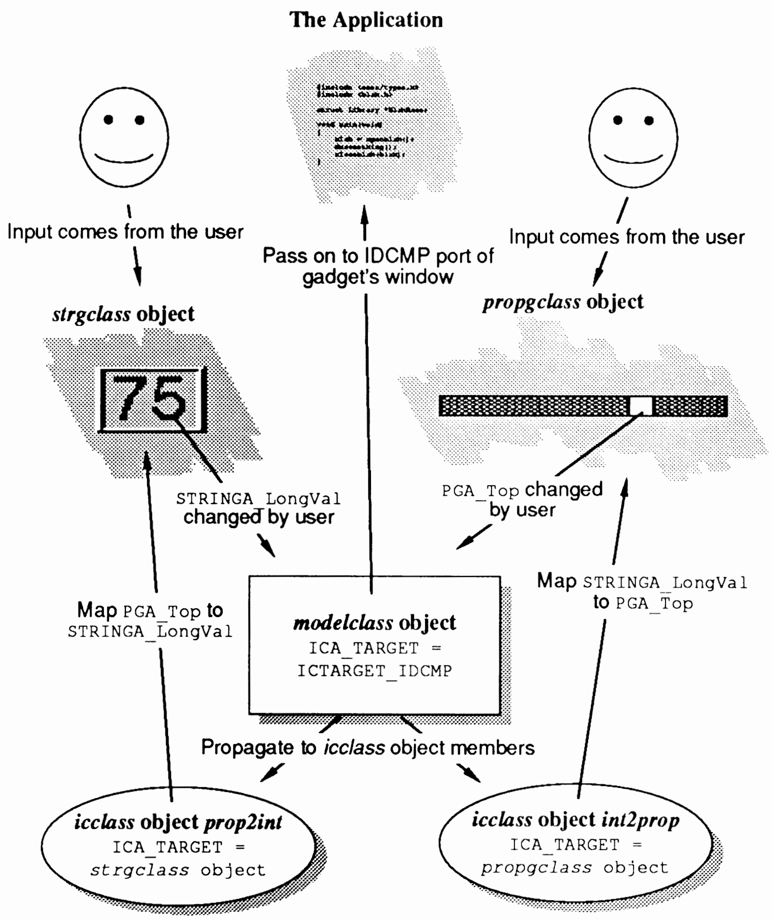


Figure 12-4: ICC Diagram

When either of these gadgets has some interim state change (caused by the user manipulating the gadgets), it sends an OM_UPDATE message to its ICA_TARGET, which in this case is the modelclass object. When this model gets the message, it does two things. It sends an IDCMP_IDCMPUPDATE to the IDCMP port of the gadget's window and it also sends OM_UPDATE messages to all of the objects in its personal list. When **int2prop** gets an OM_UPDATE message, it forwards that message to its ICA_TARGET, the prop gadget. Similarly, when **prop2int** gets an OM_UPDATE message, it forwards that message to its ICA_TARGET, the integer gadget.

Although in this case it isn't a problem, icclass and modelclass objects contain loop inhibition capabilities. If an icclass object (or modelclass object) receives an OM_UPDATE message, it forwards the message to its target. If somehow that forwarded message gets forwarded (or broadcast) back to the icclass object, the icclass object ignores the message. This prevents the possibility of an infinite OM_UPDATE loop.

Creating a Boopsi Class

So far this chapter has only hinted at what is possible with Boopsi. Its power lies in its extensibility. Boopsi grants the application programmer the power to add custom features to existing classes. If an existing class comes close to your needs, you can build on that class so it does exactly what you want. If you want a class that is unlike an existing class, you can create it.

The heart of a Boopsi class is its method *Dispatcher* function. According to the OOP metaphor, when an application wants a Boopsi object to perform a method, it sends the object a message. In reality, that object is only a data structure, so it does not have the power to do anything. When an object receives a Boopsi message, a Boopsi message structure is passed to the dispatcher of that object's class. The dispatcher examines the message and figures out what to do about it.

For example, when an application calls **SetGadgetAttrs()** on an integer gadget:

```
SetGadgetAttrs(myintegergadget, mywindow, NULL,
               STRINGA_LongVal, 75L,
               GA_ID,           2L,
               TAG_END));
```

the **SetGadgetAttrs()** function calls the strgclass dispatcher. A Boopsi dispatcher receives three arguments: a pointer to the dispatcher's **Class** (defined in <*intuition/classes.h*>), a pointer to the object that is going to perform the method, and a pointer to the Boopsi message. In this case, the **SetGadgetAttrs()** function builds an OM_SET message, finds the strgclass dispatcher, and "sends" the dispatcher the OM_SET message. **SetGadgetAttrs()** can find the dispatcher because an object contains a reference to its dispatcher.

When the dispatcher function "gets" the message, it examines the message to find out its corresponding method. In this case, the dispatcher recognizes the message as an OM_SET message and proceeds to set **myintegergadget**'s attributes.

An OM_SET message looks like this (defined in <*intuition/classusr.h*>):

```
struct opSet {
    ULONG MethodID;                    /* This will be set to OM_SET      */
    struct TagItem    *ops_AttrList;   /* A tag list containing the       */
                                       /*     attribute/value pairs of    */
                                       /*     the attributes to set.      */
    struct GadgetInfo *ops_GInfo;      /* Special information for gadgets */
}
```

The OM_SET message contains a pointer to a tag list in ops_AttrList that looks like this:

```
{STRINGA_LongVal, 75L},
{GA_ID, 2L},
{TAG_END,}
```

The strgclass dispatcher scans through this tag list and recognizes the STRINGA_LongVal attribute. The dispatcher sets myintegergadget's internal STRINGA_LongVal value to the corresponding value (75L) from the attribute/value pair.

The strgclass dispatcher continues to scan through the tag list. When it finds GA_ID, it does not process it like STRINGA_LongVal. The strgclass dispatcher's OM_SET method does not recognize the GA_ID attribute because strgclass *inherited* the GA_ID attribute from gadgetclass. To handle setting the GA_ID attribute, the strgclass dispatcher passes on the OM_SET message to its superclass's dispatcher. The strgclass dispatcher passes control to the gadgetclass dispatcher, which knows about the GA_ID attribute.

BUILDING ON EXISTING PUBLIC CLASSES

A program can create its own subclasses which build on the features of existing classes. For example, a program could create a subclass of modelclass named rkmmodelclass. Rkmmodelclass builds on modelclass by adding a new attribute called RKMMOD_CurrVal. This purpose of this attribute is simply to hold an integer value.

Because this new attribute is built into an rkmmodel object, the object could be implemented so that it exercises a certain amount of control over that value. For example, rkmmodelclass could be implemented so an rkmmodel performs bounds checking on its internal value. When an application asks an rkmmodel to set its internal RKMMOD_CurrVal, the rkmmodel makes sure the new value is not beyond a maximum value. If the new value is beyond the maximum, it sets its current value to the maximum. After the rkmmodelclass object has set its internal RKMMOD_CurrVal, it can broadcast the change on to objects in its broadcast list.

The dispatcher for rkmmodelclass does not have to do a lot of work because it inherits most of its behavior from its superclasses. The rkmmodelclass has to take care of setting aside memory for the RKMMOD_CurrVal attribute and processing any OM_SET requests to set the RKMMOD_CurrVal attribute. For any other attributes or methods, the rkmmodelclass dispatcher passes on processing to its superclass, modelclass.

Building Rkmmodelclass

So far, the theoretical class rkmmodelclass has just one attribute, RKMMOD_CurrVal. A couple of extra attributes can make it more useful. Because the rkmmodel object maintains an upper limit on its RKMMOD_CurrVal integer value, it would be useful if that upper limit was variable. Using a new attribute, RKMMOD_Limit, an application can tell a rkmmodel what its upper limit is. The rkmmodel will enforce the limit internally, so the application doesn't have to worry about it.

Another useful addition is a pulse increment and decrement for RKMMOD_CurrVal. Whenever the model receives an increment or decrement command, it increments or decrements its internal value. To make the example class simple, rkmmodelclass implements incrementing and decrementing by creating "dummy" attributes called RKMMOD_Up and RKMMOD_Down. When an rkmmodel receives an OM_SET message for one of these attributes, it increments or decrements RKMMOD_CurrVal. An rkmmodelclass object does not care what the value of the RKMMOD_Up and RKMMOD_Down attributes are, it only cares that it received an OM_UPDATE about it.

There are two pieces of data that make up this new class's instance data: the rkmmodel's current value (RKMMOD_CurrVal) and the upper limit of the rkmmodel (RKMMOD_Limit). The example class consolidates them into one structure:

```
struct RKMModData {
    ULONG currval;
    ULONG vallimit;
};
```

WRITING THE DISPATCHER

The C prototype for a Boopsi dispatcher looks like this:

```
ULONG dispatchRKMModel(Class *cl, Object *recvobject, Msg msg);
```

where **cl** points to the **Class** (defined in *<intuition/classes.h>*) of the dispatcher, **recvobject** points to the object that received the message, and **msg** is that Boopsi message. The format of the message varies according to the method. The default Boopsi message is an **Msg** (from *<intuition/classusr.h>*):

```
typedef struct {
    ULONG MethodID;
} *Msg;
```

Boopsi methods that require parameters use custom message structures. The first field of any message structure is always the method's methodID. This makes custom messages look like an **Msg**. The dispatcher looks at an incoming message's first field to tell what its method is. Rkmmodelclass objects respond to several rootclass methods:

OM_NEW
: This method creates a new rkmmodelclass object. It uses an **opSet** structure as its Boopsi message.

OM_DISPOSE
: This method tells an object to dispose of itself. It uses an **Msg** as its Boopsi message.

OM_SET
: This method tells an object to set one or more of its attribute values. It uses an **opSet** structure as its Boopsi message.

OM_UPDATE
: This method tells an object to update one or more of its attribute values. It uses an **opUpdate** structure as its Boopsi message.

OM_GET
: This method tells an object to report an attribute value. It uses an **opGet** structure as its Boopsi message.

OM_ADDTAIL
: This method tells an object to add itself to the end of an Exec list. It uses an **opAddTail** structure as its Boopsi message.

OM_REMOVE
: This method tells an object to remove itself from an Exec list. It uses an **Msg** as its Boopsi message.

OM_ADDMEMBER
: This method tells an object to add an object to its broadcast list. It uses an **opMember** structure as its Boopsi message.

OM_REMMEMBER
: This method tells an object to remove an object from its broadcast list. It uses an **opMember** structure as its Boopsi message.

OM_NOTIFY
: This method tells an object to broadcast an attribute change to its broadcast list. It uses an **opSet** structure as its Boopsi message.

Of these, rkmmodelclass has to process OM_NEW, OM_SET, OM_UPDATE, and OM_GET.

OM_NEW

The OM_NEW method returns a pointer to a newly created Boopsi object, or NULL if it failed to create the object. This method receives the following message structure (defined in *<intuition/classusr.h>*):

```
struct opSet {                    /* The OM_NEW method uses the same structure as OM_GET */
    ULONG              MethodID;
    struct TagItem    *ops_AttrList;
    struct GadgetInfo *ops_GInfo;
};
```

The ops_AttrList field contains a pointer to a **TagItem** array of attribute/value pairs. These contain the initial values of the new object's attributes. The **ops_GInfo** field is always NULL for the OM_NEW method.

Unlike other methods, when a dispatcher gets an OM_NEW message, the object pointer (**recvobject** from the **dispatchRKMModel()** prototype above) does not point to an object. It doesn't make sense for **recvobject** to point to an object because the idea is to create a new object, not act on an existing one.

The pointer normally used to pass a Boopsi object is instead used to pass the address of the object's ''true class''. An object's true class is the class of which the object is an instance.

The first thing the dispatcher does when it processes an OM_NEW message is pass the OM_NEW message on to its superclass's dispatcher. It does this using the *amiga.lib* function **DoSuperMethodA()**:

```
ULONG DoSuperMethodA(Class *cl, Object *trueclass, Msg msg);
```

Each dispatcher passes control to its superclass. Eventually the message will arrive at the rootclass dispatcher. The OM_NEW method in the rootclass dispatcher looks at the object's true class (**trueclass** from the prototype) to find out which class dispatcher is trying to create a new object. Note that **trueclass** is not necessarily the same as the current dispatcher's class (**cl** from the **dispatchRKMModel()** prototype above), although this would be the case if the object's true class is a subclass of the current dispatcher's class.

The rootclass dispatcher uses the true class to find out how much memory to allocate for the object's instance data. Each class keeps a record of how much memory its local instance data requires. The rootclass dispatcher also looks at each class between the true class and rootclass to find out much memory the local instance data for those classes require. The rootclass dispatcher totals the amount of local instance data memory needed by the true class and each of its superclasses and allocates that much memory.

If all goes well, the rootclass dispatcher increments a private field in the true class that keeps track of how many instances of the true class there currently are. It then returns a pointer to the newly created object and passes control back to the subclass dispatcher that called it, which is icclass in the case of rkmmodelclass. If there was a problem, the rootclass dispatcher does not increment the object count and passes back a NULL.

When the rootclass dispatcher returns, the *icclass dispatcher* regains control from **DoSuperMethodA()**. **DoSuperMethodA()** will return either a pointer to the new object or else it returns NULL if there was an error. Although the rootclass dispatcher allocated all the memory the object needs, it only initialized the instance data local to rootclass. Now it's the icclass dispatcher's turn to do some work. It has to initialize the instance data that is local to icclass.

A dispatcher finds its local instance data by using the **INST_DATA()** macro (defined in *<intuition/classes.h>*):

```
void *INST_DATA(Class *localclass, Object *object);
```

INST_DATA() takes two arguments, a pointer to a class and a pointer to the object. The **INST_DATA()** macro returns a pointer to the instance data local to **localclass**. When the icclass dispatcher was called, it received three arguments, one of which was a pointer to the local class (icclass). The icclass dispatcher passes this pointer and the new object pointer it got from **DoSuperMethodA()** to **INST_DATA()** to get a pointer to the instance data local to icclass.

After initializing its local instance data, the icclass dispatcher passes control back to the modelclass dispatcher, which in turn, initializes the instance data local to modelclass. Finally, the rkmmodelclass dispatcher regains control and now has to take care of its local instance data.

To find its local instance data, the rkmmodelclass dispatcher needs a pointer to its **Class** and a pointer to the new object. The dispatcher function gets its **Class** pointer as its first argument (**cl** from the **dispatchRKMModel()** prototype above). It gets the new object pointer as the return value from **DoSuperMethodA()**. In this case, **INST_DATA()** returns a pointer to an **RKMModData** structure.

Now the dispatcher has to initialize its local instance data. It has to scan through the tag list passed in the OM_NEW message looking for initial values for the RKMMOD_CurrVal and RKMMOD_Limit attributes. As an alternative, the dispatcher's OM_NEW method can use its OM_SET method to handle initializing these ''settable'' attributes.

Finally, the dispatcher can return. When the dispatcher returns from an OM_NEW method, it returns a pointer to the new object.

If the OM_NEW method fails, it should tell the partially initialized object it got from its superclass's dispatcher to dispose of itself (using OM_DISPOSE) and return NULL.

OM_SET/OM_UPDATE

For the OM_SET message, the rkmmodelclass dispatcher steps through the attribute/value pairs passed to it in the OM_SET message looking for the local attributes (see OM_NEW for the OM_SET message structure). The RKMMOD_Limit attribute is easy to process. Just find it and record the value in the local **RKMModData.vallimit** field.

Because the function of the rkmmodelclass's OM_SET and OM_UPDATE methods are almost identical, the rkmmodelclass dispatcher handles them as the same case. The only difference is that, because the OM_UPDATE message comes from another Boopsi object, the OM_UPDATE method can report on transitory state changes of an attribute. For example, when the user slides a Boopsi prop gadget, that prop gadget sends out an interim OM_UPDATE message for every interim value of PGA_Top. When the user lets go of the prop gadget, the gadget sends out a final OM_UPDATE message. The OM_UPDATE message is almost identical to the OM_SET message:

```
#define OPUF_INTERIM    (1<<0)
struct opUpdate {                          /* the OM_NOTIFY method uses the same structure */
    ULONG              MethodID;
    struct TagItem     *opu_AttrList;
    struct GadgetInfo  *opu_GInfo;
    ULONG              opu_Flags;          /* The extra field */
};
```

A dispatcher can tell the difference between an interim and final OM_UPDATE message because the OM_UPDATE message has an extra field on it for flags. If the low order bit (the OPUF_INTERIM bit) is set, this is an interim OM_UPDATE message. The interim flag is useful to a class that wants to ignore any transitory messages, processing only final attribute values. Because rkmmodelclass wants to process all changes to its attributes, it processes all OM_UPDATE messages.

The RKMMOD_CurrVal attribute is a little more complicated to process. The dispatcher has to make sure the new current value is within the limits set by RKMMOD_Limit, then record that new value in the local **RKMModData.currval** field. Because other objects need to hear about changes to RKMMOD_CurrVal, the dispatcher has to send a *notification* request. It does this by sending itself an OM_NOTIFY message. The OM_NOTIFY message tells an object to notify its targets (its ICA_TARGET and the objects in its broadcast list) about an attribute change. The OM_NOTIFY method does this by sending OM_UPDATE messages to all of an object's targets.

The rkmmodelclass dispatcher does not handle the OM_NOTIFY message itself. It inherits this method from modelclass, so the rkmmodelclass dispatcher passes OM_NOTIFY messages on to its superclass. To notify its targets, the rkmmodelclass dispatcher has to construct an OM_NOTIFY message. The OM_NOTIFY method uses the same message structure as OM_UPDATE. Using the stack-based version of **DoSuperMethodA()**, **DoSuperMethod()**, the dispatcher can build an OM_NOTIFY message on the stack:

```
 . . .

struct TagItem tt[2];
struct opUpdate *msg;
 . . .

tt[0].ti_Tag  = RKMMOD_CurrVal;  /* make a tag list.  */
tt[0].ti_Data = mmd->currval;
tt[1].ti_Tag  = TAG_END;

DoSuperMethod(cl, o, OM_NOTIFY, tt, msg->opu__GInfo,
              ((msg->MethodID == OM_UPDATE) ? (msg->opu_Flags) : 0L));
 . . .
```

Because the OM_NOTIFY needs a tag list of attributes about which to issue updates, the dispatcher builds a tag list containing just the RKMMOD_CurrVal tag and its new value. The dispatcher doesn't use the tag list passed to it in the OM_UPDATE/OM_NOTIFY message because that list can contain many other attributes besides RKMMOD_CurrVal.

The **msg** variable in the **DoSuperMethod()** call above is the OM_SET or OM_UPDATE message that was passed to the dispatcher. The dispatcher uses that structure to find a pointer to the **GadgetInfo** structure that the OM_NOTIFY message requires. The **GadgetInfo** structure comes from Intuition and contains information that Boopsi gadgets need to render themselves. For the moment, don't worry about what the **GadgetInfo** structure actually does, just pass it on. The targets of an rkmmodel will probably need it.

Notice that the dispatcher has to test to see if the message is an OM_SET or OM_UPDATE so it can account for the **opu_Flags** field at the end of the OM_UPDATE message.

Processing the RKMMOD_Up and RKMMOD_Down attributes is similar to the RKMMOD_CurrVal attribute. When the dispatcher sees one of these, it has to increment or decrement the local **RKMModData.currval**, making sure **RKMModData.currval** is within limits. The dispatcher then sends an OM_NOTIFY message to the superclass about the change to **RKMModData.currval.**

The return value from the dispatcher's OM_SET method depends on the what effect the attribute change has to the visual state of the objects in the rkmmodel's broadcast list. If an attribute change will not affect the visual state of the rkmmodel's objects, the OM_SET method returns zero. If the attribute change could trigger a change to the rkmmodel's objects, it returns something besides zero. For example, the rkmmodelclass OM_SET method returns 1L if an rkmmodel's RKMMOD_CurrVal, RKMMOD_Up, or RKMMOD_Down attribute is changed.

At some point the rkmmodelclass dispatcher has to allow its superclasses to process these attributes it inherits. Normally a dispatcher lets the superclass process its attributes before attempting to process any local attributes. The rkmmodelclass dispatcher does this by passing on the OM_SET or OM_UPDATE message using **DoSuperMethodA()** (inheritance at work!). As an alternative, the dispatcher can use the *amiga.lib* function **SetSuperAttrs()**. See the *amiga.lib* Autodocs for more details on this function.

OM_GET

The rkmmodel only has one "gettable" attribute: RKMMOD_CurrVal, which makes processing it easy. The OM_GET message looks like this (defined in *<intuition/classusr.h>*):

```
struct opGet {
    ULONG MethodID;      /* OM_GET */
    ULONG opg_AttrID;    /* The attribute to retrieve */
    ULONG *opg_Storage;  /* a place to put the attribute's value */
};
```

When the rkmmodelclass dispatcher receives an OM_GET message with an **opg_AttrID** equal to RKMMOD_CurrVal, it copies the current value (**RKMModData.currval**) to the memory location **opg_Storage** points to and returns a value of TRUE. The TRUE indicates that there was no error. If **opg_AttrID** is not RKMMOD_CurrVal, the dispatcher should let its superclass handle this message.

The rkmmodelclass dispatcher can take advantage of the fact that the only "gettable" attribute available to an rkmmodel is RKMMOD_CurrVal (the attributes defined by modelclass and icclass are not gettable—see the *Boopsi Class Reference* in the Appendix B of this manual for more details on which attributes are "settable", "gettable", etc.). If **opg_AttrID** is not RKMMOD_CurrVal, the rkmmodelclass dispatcher can return FALSE, indicating that the attribute was not "gettable".

If the rkmmodelclass dispatcher comes across any other messages besides OM_NEW, OM_SET, OM_UPDATE, and OM_GET message, it blindly passes them on to its superclass for processing.

Making the New Class

The Intuition function **MakeClass()** creates a new Boopsi class:

```
Class *MakeClass(UBYTE *newclassID, UBYTE *pubsuperclassID, Class *privsuperclass,
                UWORD instancesize, ULONG flags);
```

If the new class is going to be public, **newclassID** is a string naming the new class. If the new class is private, this field is NULL. The next two fields tell **MakeClass()** where to find the new class's superclass. If the superclass is public, **pubsuperclassID** points to a string naming that public superclass and the **privsuperclass** pointer is NULL. If the superclass is private, **privsuperclass** points to that superclass's **Class** structure and **pubsuperclassID** is NULL. The size of the new class's local instance data is **instancesize**. The last parameter, **flags**, is for future enhancement. For now, make this zero.

If it is successful, **MakeClass()** returns a pointer to the new class, otherwise it returns NULL. When **MakeClass()** is successful, it also takes measures to make sure no one can "close" the new class's superclass (using **FreeClass()**). It does this by incrementing a private field of the superclass that keeps track of how many subclasses the superclass currently has.

After successfully creating a class, an application has to tell the class where its dispatcher is. The **Class** pointer (defined in *<intuition/classes.h>*) returned by **MakeClass()** contains a **Hook** structure called **cl_Dispatcher**, which is used to call the dispatcher. The application has to initialize this hook:

```
myclass->cl_Dispatcher.h_Entry = HookEntry; /* <------ HookEntry() is defined in amiga.lib */
myclass->cl_Dispatcher.h_SubEntry = dispatchRKMModel;
```

The **h_Entry** field points to a function in *amiga.lib* that copies the function arguments to where the dispatcher expects them. See the *Callback Hooks* section of the "Utility Library" chapter of this manual for more details.

To make a class public instead of private, an application has to call **AddClass()** in addition to giving the class a name in **MakeClass()**. **AddClass()** takes one argument, a pointer to a valid **Class** structure that has been initialized as a public class by **MakeClass()**. To remove a public class added to the system with **AddClass()**, pass the public class pointer to **RemoveClass()**. See the Intuition Autodocs for more details on **AddClass()** and **RemoveClass()**.

RKMModel.c

The following code, *RKMModel.c*, makes up an initialization function and the dispatcher function for a private class informally called rkmmodelclass.

```
;/* RKMModel.c - A simple custom modelclass subclass.
LC -cfist -b1 -y -v -j73 rkmmodel.c
quit ;*/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/classes.h>
#include <intuition/classusr.h>
#include <intuition/imageclass.h>
#include <intuition/gadgetclass.h>
#include <intuition/cghooks.h>
#include <intuition/icclass.h>
#include <utility/tagitem.h>
#include <utility/hooks.h>
#include <clib/intuition_protos.h>
#include <clib/utility_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>

extern struct Library *IntuitionBase, *UtilityBase;

/*************************************************************************************************/
/****************  The attributes defined by this class  *****************************************/
/*************************************************************************************************/
#define RKMMOD_CurrVal   (TAG_USER + 1) /* This attribute is the current value of the model.*******/
                                        /**********************************************************/
#define RKMMOD_Up        (TAG_USER + 2) /* These two are fake attributes that rkmmodelclass *******/
#define RKMMOD_Down      (TAG_USER + 3) /* uses as pulse values to increment/decrement the  *******/
                                        /* rkmmodel's RKMMOD_CurrVal attribute.             *******/
                                        /**********************************************************/
#define RKMMOD_Limit     (TAG_USER + 4) /* This attribute contains the upper bound of the   *******/
                                        /* rkmmodel's RKMMOD_CurrVal.  The rkmmodel has a   *******/
                                        /* static lower bound of zero.                      *******/
/*************************************************************************************************/

#define DEFAULTVALLIMIT 100L /* If the programmer doesn't set          */
                             /* RKMMOD_Limit, it defaults to this.     */
```

```
struct RKMModData {
    ULONG currval;      /* The instance data for this class.            */
    ULONG vallimit;
};

/***************************************************************************************/
/*************************      The functions in this module     ***********************/
/***************************************************************************************/
void    geta4(void);                                                   /***************/
Class  *initRKMModClass(void);                                         /***************/
BOOL    freeRKMModClass(Class *);                                      /***************/
ULONG   dispatchRKMModel(Class *, Object *, Msg);                      /***************/
void    NotifyCurrVal(Class *, Object *, struct opUpdate *, struct RKMModData *); /***************/
/***************************************************************************************/

/***************************************************************************************/
/********************************   Initialize the class    ***************************/
/***************************************************************************************/
Class   *initRKMModClass(void)                /* Make the class and set     */
{                                             /* up the dispatcher's hook.  */
    Class *cl;
    extern ULONG HookEntry();  /*     <-------   defined in amiga.lib.     */

    if ( cl =  MakeClass( NULL,
                "modelclass", NULL,
                sizeof ( struct RKMModData ),
                0 ))
    {
        cl->cl_Dispatcher.h_Entry = HookEntry;           /* initialize the  */
        cl->cl_Dispatcher.h_SubEntry = dispatchRKMModel; /* cl_Dispatcher   */
                                                         /* Hook.           */
    }
    return ( cl );
}

/***************************************************************************************/
/********************************      Free the class     *****************************/
/***************************************************************************************/
BOOL freeRKMModClass( Class *cl )
{
    return (FreeClass(cl));
}

/***************************************************************************************/
/********************************     The class Dispatcher    *************************/
/***************************************************************************************/
ULONG dispatchRKMModel(Class *cl, Object *o, Msg msg)
{
    struct RKMModData *mmd;
    APTR retval = NULL;  /* A generic return value used by this class's methods.  The          */
                         /* meaning of this field depends on the method.  For example,        */
                         /* OM_GET uses this a a boolean return value, while OM_NEW           */
                         /* uses it as a pointer to the new object.                           */
    geta4();      /* SAS/C and Manx function - makes sure A4 contains global data pointer.     */

    switch (msg->MethodID)
    {
        case OM_NEW:     /* Pass message onto superclass first so it can set aside the memory   */
                         /* for the object and take care of superclass instance data.          */
            if (retval = (APTR)DoSuperMethodA(cl, o, msg))
            {            /*******************************************************************/
                         /* For the OM_NEW method, the object pointer passed to the dispatcher */
                         /* does not point to an object (how could it? The object doesn't exist */
                         /* yet.)  DoSuperMethodA() returns a pointer to a newly created       */
                         /* object. INST_DATA() is a macro defined in <intuition/classes.h>   */
                         /* that returns a pointer to the object's instance data that is local */
                         /* to this class. For example, the instance data local to this class  */
                         /* is the RKMModData structure defined above.                         */
                         /*******************************************************************/
                mmd = INST_DATA(cl, retval);
                mmd->currval = GetTagData(RKMMOD_CurrVal, 0L, /* initialize object's attributes. */
                                          ((struct opSet *)msg)->ops_AttrList);
                mmd->vallimit = GetTagData(RKMMOD_Limit, DEFAULTVALLIMIT,
                                           ((struct opSet *)msg)->ops_AttrList);
            }
            break;
```

```
        case OM_SET:
        case OM_UPDATE:
            mmd = INST_DATA(cl, o);
            DoSuperMethodA(cl, o, msg);    /* Let the superclasses set their attributes first.    */
            {
                struct TagItem *tstate, *ti;    /* grab some temp variables off of the stack.    */

                ti = ((struct opSet *)msg)->ops_AttrList;
                tstate = ti;

                        /* Step through all of the attribute/value pairs in the list.  Use the   */
                        /* utility.library tag functions to do this so they can properly process */
                        /* special tag IDs like TAG_SKIP, TAG_IGNORE, etc.                       */

                while (ti = NextTagItem(&tstate))     /* Step through all of the attribute/value  */
                {                  /* pairs in the list. Use the utility.library tag functions    */
                                   /* to do this so they can properly process special tag IDs     */
                                   /* like TAG_SKIP, TAG_IGNORE, etc.                             */
                    switch (ti->ti_Tag)
                    {
                        case RKMMOD_CurrVal:
                            if ((ti->ti_Data) > mmd->vallimit) ti->ti_Data =
                                    mmd->vallimit;
                            mmd->currval = ti->ti_Data;
                            NotifyCurrVal(cl, o, msg, mmd);
                            retval = (APTR)1L;  /* Changing RKMMOD_CurrVal can cause a visual    */
                            break;              /* change to gadgets in the rkmmodel's broadcast */
                                                /* list.  The rkmmodel has to tell the applica-  */
                                                /* tion by returning a value besides zero.       */
                        case RKMMOD_Up:
                            mmd->currval++;

                                 /* Make sure the current value is not greater than value limit. */
                            if ((mmd->currval) > mmd->vallimit) mmd->currval = mmd->vallimit;
                            NotifyCurrVal(cl, o, msg, mmd);
                            retval = (APTR)1L;  /* Changing RKMMOD_Up can cause a visual         */
                            break;              /* change to gadgets in the rkmmodel's broadcast */
                                                /* list.  The rkmmodel has to tell the applica-  */
                                                /* tion by returning a value besides zero.       */
                        case RKMMOD_Down:
                            mmd->currval--;
                                    /* Make sure currval didn't go negative. */
                            if ((LONG)(mmd->currval) == -1L)
                                mmd->currval = 0L;
                            NotifyCurrVal(cl, o, msg, mmd);
                            retval = (APTR)1L;  /* Changing RKMMOD_Down can cause a visual       */
                            break;              /* change to gadgets in the rkmmodel's broadcast */
                                                /* list.  The rkmmodel has to tell the applica-  */
                                                /* tion by returning a value besides zero.       */
                        case RKMMOD_Limit:
                            mmd->vallimit = ti->ti_Data; /* Set the limit.  Note that this does  */
                            break;                       /* not do bounds checking on the        */
                                                         /* current RKMModData.currval value.    */
                    }
                }
            }
            break;
        case OM_GET:                         /* The only attribute that is "gettable" in this      */
            mmd = INST_DATA(cl, o);          /* class or its superclasses is RKMMOD_CurrVal.       */
            if ((((struct opGet *)msg)->opg_AttrID) == RKMMOD_CurrVal)
            {
                *(((struct opGet *)msg)->opg_Storage) = mmd->currval;
                retval = (Object *)TRUE;
            }
            else retval = (APTR)DoSuperMethodA(cl, o, msg);
            break;
        default:          /* rkmmodelclass does not recognize the methodID, so let the superclass's */
                          /* dispatcher take a look at it.                                          */
            retval = (APTR)DoSuperMethodA(cl, o, msg);
            break;
    }
    return((ULONG)retval);
}
```

```
void NotifyCurrVal(Class *cl, Object *o, struct opUpdate *msg, struct RKMModData *mmd)
{
    struct TagItem tt[2];

    tt[0].ti_Tag = RKMMOD_CurrVal; /* make a tag list.  */
    tt[0].ti_Data = mmd->currval;
    tt[1].ti_Tag = TAG_DONE;
                              /* If the RKMMOD_CurrVal changes, we want everyone to know about */
    DoSuperMethod(cl, o,      /* it. Theoretically, the class is supposed to send itself a     */
            OM_NOTIFY,        /* OM_NOTIFY message. Because this class lets its superclass     */
            tt,               /* handle the OM_NOTIFY message, it skips the middleman and      */
            msg->opu_GInfo,   /* sends the OM_NOTIFY directly to its superclass.               */

            ((msg->MethodID == OM_UPDATE) ? (msg->opu_Flags) : 0L)); /* If this is an OM_UPDATE */
                              /* method, make sure the part the OM_UPDATE message adds to the  */
                              /* OM_SET message gets added.  That lets the dispatcher handle   */
}                             /* OM_UPDATE and OM_SET in the same case.                        */
```

Below is a diagram showing how an application could use an rkmmodelclass object:

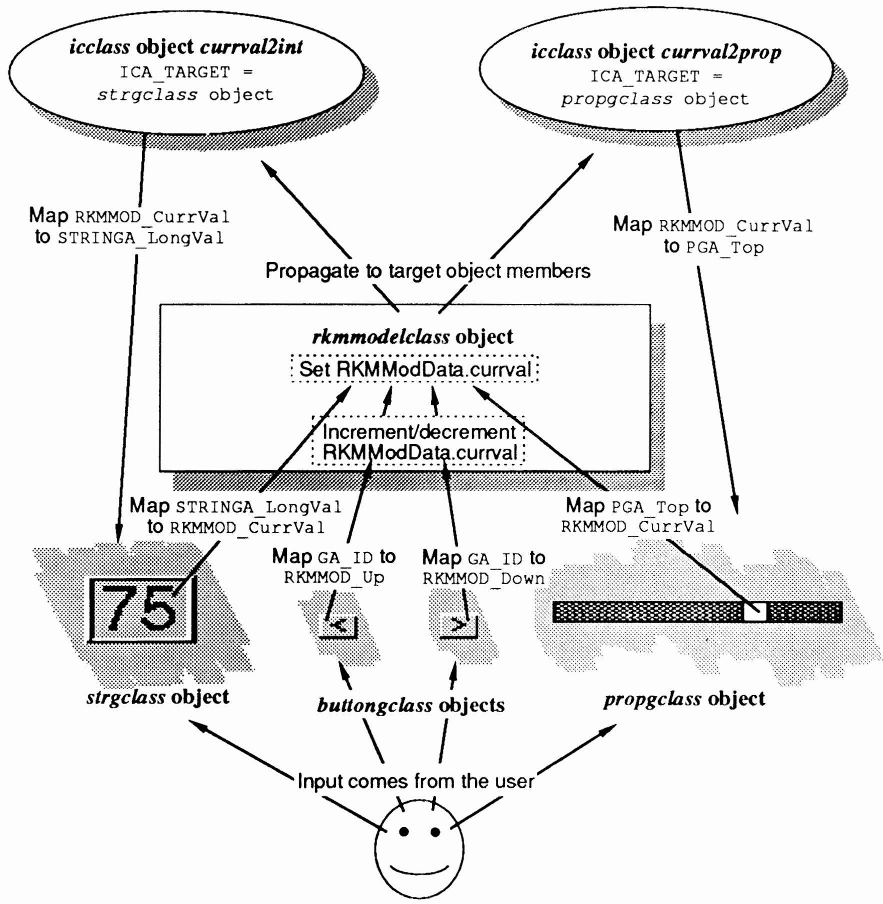


Figure 12-5: Rkmmodelclass Object Diagram

In this diagram, the application uses buttongclass Boopsi gadgets to send the rkmmodelclass the RKMMOD_Up and RKMMOD_Down attribute pulses.

The example takes advantage of an odd feature of buttongclass. When the user clicks on a buttongclass gadget, it sends an OM_UPDATE to its ICA_TARGET, even though no Boopsi attribute of buttongclass has changed. It does this because it's a convenient way to report button clicks.

Whenever a gadget sends a notification, the list of attribute/value pairs in the OM_NOTIFY message always contains the gadget's GA_ID. This is an easy way for the button to inform its target of its ID so the target knows which gadget sent the OM_UPDATE message. When a buttongclass sends a notification because of a button click, it only sends out an OM_UPDATE about its GA_ID because none of its attributes changed.

When the user clicks one of the buttons in the rkmmodelclass diagram, the button uses an ICA_MAP to map its GA_ID to one of the "dummy" pulse attributes, RKMMOD_Up and RKMMOD_Down. When the rkmmodel receives the OM_UPDATE message about RKMMOD_Up or RKMMOD_Down, it increments or decrements its internal value.

There is one more important thing to note about rkmmodelclass. Looking at the rkmmodelclass Object diagram above, an rkmmodel's RKMMOD_CurrVal changes because it received an OM_UPDATE message from one of its gadgets. RKMMOD_CurrVal can also change if the application explicitly set RKMMOD_CurrVal using **SetAttrs()** or **SetGadgetAttrs()**.

The primary difference between the OM_SET message that **SetAttrs()** sends and the OM_SET message that **SetGadgetAttrs()** sends is that **SetAttrs()** passes a NULL in **opSet.ops_GInfo** instead of a **GadgetInfo** pointer. This doesn't present a problem for the rkmmodel object, because it doesn't use the **GadgetInfo** structure. The problem is that when the rkmmodel notifies its targets, some of which are gadgets, they can't update their visual state because they need a **GadgetInfo** to render themselves. For this reason, the rkmmodelclass dispatcher returns a positive non-zero value when an attribute change occurs that could cause a change in the visual state of any objects in its broadcast list. An application that uses rkmmodelclass must test the return value when calling **SetAttrs()** on an rkmmodelclass object to tell if the attribute change requires a visual refresh of the gadgets (see the Intuition Autodocs for **RefreshGadgets()**).

> *Boopsi Dispatchers Can Execute on Intuition's Context.* Notice that the gadgets in the figure above send OM_UPDATE messages to the rkmmodel when the user manipulates them. Because Intuition handles the user input that triggers the OM_UPDATE messages, Intuition itself is sending the OM_UPDATE messages. This means the rkmmodelclass dispatcher must be able to run on Intuition's context, which puts some limitations on what the dispatcher is permitted to do: it can't use dos.library, it can't wait on application signals or message ports, and it can't call any Intuition functions which might wait on Intuition.

Although rkmmodelclass serves as an example of a class, it leaves a little to be desired in a real-world implementation. To create the "prop-integer-up/down" super gadget from the diagram above, the application has to create, initialize, and link nine Boopsi objects, which is tedious, especially if the application needs several of these super gadgets. Ideally, all these functions would be rolled into some subclass of gadgetclass. If there were such a class, an application would only have to create one instance of this subclass to get such a gadget. When the subclass received an OM_NEW message, it would take care of creating, initializing, and linking all of the Boopsi objects that make up the whole super gadget.

WHITE BOXES—THE TRANSPARENT BASE CLASSES

Boopsi gadgets and images were designed to be backwards compatible with the old Intuition **Gadgets** and **Images**, so as part of their instance data, both types of objects have the old Intuition structures built into them. When **NewObject()** creates a new gadget or image object, the pointer it returns points to the object's embedded **Gadget** or **Image** corresponding structure. Because Intuition can tell the difference between Boopsi images and gadgets and the original images and gadgets, applications can use Boopsi images and gadgets interchangeably with the older Intuition entities.

Although normally considered a "programming sin", in some cases it is legal for class dispatchers to directly manipulate some internal fields of certain Boopsi objects. For compatibility reasons, a Boopsi image or gadget object contains an actual **Image** or **Gadget** structure. These objects are instances of the *Transparent Base Classes*, imageclass and gadgetclass.

To change an attribute of a Boopsi object, you normally invoke the set method, OM_SET. The Intuition functions **SetAttrs()** and **SetGadgetAttrs()** invoke this method. A Boopsi class is informed of any attribute change at that time, allowing it to react to this change. The reaction can include validating the changed attribute, changing other attributes to match, or informing other objects of the change. That is the inherent advantage of using function calls to change attributes.

When using conventional images and gadgets, you generally modify the structure's fields directly. This operation is very fast. For conventional images and gadgets, there is no class that needs to know about the changes, so there is no problem. However, this is untrue of Boopsi images and gadgets. Although directly modifying the Boopsi object's internal structure would provide a performance increase over using the Boopsi OM_SET mechanism, altering a Boopsi object's internal structure directly will not give the class the opportunity to react to any structure changes. This violates the Boopsi concept, and therefore cannot be done in general.

In order to provide a balance between the flexibility of function-access and the performance of direct-access, the transparent base classes imageclass and gadgetclass do not depend on being informed of changes to certain fields in the internal **Image** and **Gadget** structures. This means that it is OK for the dispatchers of direct subclasses of imageclass and gadgetclass to modify specific fields of Boopsi images or gadgets. Applications and indirect subclass dispatchers of imageclass or gadgetclass may not modify those fields, since their parent classes may depend on hearing about changes to these fields, which the **SetAttrs()** call (or a similar function) provides.

For dispatchers of direct subclasses of imageclass, the following are the only fields of the **Image** structure that are alterable by application programs:

LeftEdge	Width	ImageData
TopEdge	Height	PlanePick
PlaneOnOff		

For dispatchers of direct subclasses of gadgetclass, the following are the only fields of the **Gadget** structure that are alterable by application programs:

LeftEdge	Flags	GadgetText
TopEdge	GadgetType	SpecialInfo
Width	GadgetRender	Activation
Height	SelectRender	

Under no circumstances may an application or an indirect subclass modify one of these fields, even if the subclass knows the superclasses do not depend on notification for this field. This is the only way to preserve the possibility for future enhancements to that superclass. Note that these fields are not alterable while the gadget or image object is in use (for example, when it is attached to a window).

Boopsi Gadgets

One of the major enhancements to Intuition for Release 2 is the implementation of customizable Boopsi gadgets. Boopsi gadgets are not limited by dependencies upon Intuition **Image** and **Gadget** structures. Unlike Release 1.3 gadgets, which were handled exclusively by Intuition, Boopsi gadgets handle their own rendering and their own user input.

Since Boopsi gadgets draw themselves, there is almost no restriction on what they can look like. A Boopsi gadget can use graphics.library **RastPort** drawing functions to draw vector-based imagery which the gadget can scale to any dimension. Instead of just a two-state Boolean gadget, a Boopsi gadget can have any number of states, each of which has its own imagery. If a programmer wanted to he could even make a Boopsi gadget that uses the animation system to render itself.

Because Boopsi gadgets handle their own input, they see all the user's input, which the gadget is free to interpret. While the user has a Boopsi gadget selected, the gadget can track mouse moves, process mouse and keyboard key presses, or watch the timer events.

The power of a Boopsi gadget is not limited to its ability to handle its own rendering and user input. Boopsi gadgets are also Boopsi objects so the gain all the benefits Boopsi provides. This means all Boopsi gadgets inherit the methods and attributes from their superclasses. Boopsi gadgets can use Boopsi images to take care of rendering their imagery. A Boopsi gadget could be a "composite" gadget that is composed of several Boopsi gadgets, images, and models.

THE BOOPSI GADGET METHODS

Intuition drives a Boopsi gadget by sending it Boopsi messages. Intuition uses a series of five Boopsi methods:

GM_RENDER	This method tells the gadget to render itself.
GM_HITTEST	This method asks a gadget whether it has been "hit" by a mouse click.
GM_GOACTIVE	This method asks a gadget if it wants to be the active gadget.
GM_HANDLEINPUT	This method passes a gadget an input event.
GM_GOINACTIVE	This method tells a gadget that it is no longer active.

The formats of each of these Boopsi messages differ, but they all have two things in common. Like all Boopsi messages, each starts with their respective method ID. For each of these methods, the method ID field is followed by a pointer to a **GadgetInfo** structure (defined in *<intuition/cghooks.h>*). The **GadgetInfo** structure contains information about the display on which the gadget needs to render itself:

```
struct GadgetInfo {
    struct Screen              *gi_Screen;
    struct Window              *gi_Window;     /* null for screen gadgets */
    struct Requester           *gi_Requester;  /* null if not GTYP_REQGADGET */

    /* rendering information: don't use these without cloning/locking.
     * Official way is to call ObtainGIRPort()
     */
    struct RastPort            *gi_RastPort;
    struct Layer               *gi_Layer;

    /* copy of dimensions of screen/window/g00/req(/group)
     * that gadget resides in.  Left/Top of this box is
     * offset from window mouse coordinates to gadget coordinates
     *  screen gadgets:                0,0 (from screen coords)
     *  window gadgets (no g00):       0,0
     *  GTYP_GZZGADGETs (borderlayer): 0,0
```

```
 *  GZZ innerlayer gadget:          borderleft, bordertop
 *  Requester gadgets:              reqleft, reqtop
 */
struct IBox              gi_Domain;

/* these are the pens for the window or screen      */
struct {
    UBYTE   DetailPen;
    UBYTE   BlockPen;
}                        gi_Pens;

/* the Detail and Block pens in gi_DrInfo->dri_Pens[] are
 * for the screen.  Use the above for window-sensitive colors.
 */
struct DrawInfo          *gi_DrInfo;

/* reserved space: this structure is extensible
 * anyway, but using these saves some recompilation
 */
ULONG                    gi_Reserved[6];
};
```

All the fields in this structure are read only.

Although this structure contains a pointer to the gadget's **RastPort** structure, applications should not use it for rendering. Instead, use the intuition.library function **ObtainGIRPort()** to obtain a copy of the **GadgetInfo**'s **RastPort**. When the gadget is finished with this **RastPort**, it should call **ReleaseGIRPort()** to relinquish the **RastPort**.

GM_RENDER

Every time Intuition feels it is necessary to redraw a Boopsi gadget, it sends a gadget a GM_RENDER message. The GM_RENDER message (defined in <*intuition/gadgetclass.h*>) tells a gadget to render itself:

```
struct gpRender
{
    ULONG             MethodID;    /* GM_RENDER */
    struct GadgetInfo *gpr_GInfo;
    struct RastPort   *gpr_RPort;  /* all ready for use */
    LONG              gpr_Redraw;  /* might be a "highlight pass" */
};
```

Some events that cause Intuition to send a GM_RENDER are: an application passed the gadget to **OpenWindow()**, the user moved or resized a gadget's window, or an application explicitly asked Intuition to refresh some gadgets.

The GM_RENDER message contains a pointer to the gadget's **RastPort** so the GM_RENDER method does not have to extract it from the **gpr_GInfo GadgetInfo** structure using **ObtainGIRPort()**). The gadget renders itself according to how much imagery it needs to replace. The **gpr_Redraw** field contains one of three values:

GREDRAW_REDRAW Redraw the entire gadget.

GREDRAW_UPDATE The user has manipulated the gadget, causing a change to its imagery. Update only that part of the gadget's imagery that is effected by the user manipulating the gadget (for example, the knob and scrolling field of the prop gadget).

GREDRAW_TOGGLE If this gadget supports it, toggle to or from the highlighting imagery.

Intuition is not the only entity that calls this method. The gadget's other methods may call this method to render the gadget when it goes through state changes. For example, as a prop gadget is following the mouse from the gadget's GM_HANDLEINPUT method, the gadget could send itself GM_RENDER messages, telling itself to update its imagery according to where the mouse has moved.

GM_HITTEST

When Intuition gets a left mouse button click in a window, one of the things it does is check through the window's list of gadgets to see if that click was inside the bounds of a gadget's **Gadget** structure (using the **LeftEdge, TopEdge, Width,** and **Height** fields). If it was (and that gadget is a Boopsi gadget), Intuition sends that gadget a GM_HITTEST message (defined in *<intuition/gadgetclass.h>*):

```
struct gpHitTest
{
    ULONG             MethodID;     /* GM_HITTEST   */
    struct GadgetInfo *gpht_GInfo;
    struct
    {
        WORD X;                     /* Is this point inside of the gadget? */
        WORD Y;
    } gpht_Mouse;
};
```

This message contains the coordinates of the mouse click. These coordinates are relative to the upper-left of the gadget (**LeftEdge, TopEdge**).

Because Intuition can only tell if the user clicked inside gadget's "bounding box", Intuition only knows that the click was close to the gadget. Intuition uses the GM_HITTEST to ask the gadget if the click was really inside the gadget. The gadget returns GMR_GADGETHIT (defined in *<intuition/gadgetclass.h>*) to tell Intuition that the user hit it, otherwise it returns zero. This method allows a gadget to be any shape or pattern, rather than just rectangular.

GM_GOACTIVE/GM_HANDLEINPUT

If a gadget returns GMR_GADGETHIT, Intuition will send it a GM_GOACTIVE message (defined in *<intuition/gadgetclass.h>*):

```
struct gpInput /* Used by GM_GOACTIVE and GM_HANDLEINPUT */
{
    ULONG             MethodID;
    struct GadgetInfo *gpi_GInfo;
    struct InputEvent *gpi_IEvent;      /* The input event that triggered this method
                                         * (for GM_GOACTIVE, this can be NULL) */
    LONG              *gpi_Termination; /* For GADGETUP IntuiMessage.Code */
    struct
    {
        WORD X;                         /* Mouse position relative to upper         */
        WORD Y;                         /* left corner of gadget (LeftEdge, TopEdge) */
    } gpi_Mouse;
};
```

The GM_GOACTIVE message gives a gadget the opportunity to become the active gadget. The active gadget is the gadget that is currently receiving user input. Under normal conditions, only one gadget can be the active gadget (it is possible to have more than one active gadget using a groupgclass object—See the *Boopsi Class Reference* in the Appendix B of this manual for more details).

While a gadget is active, Intuition sends it GM_HANDLEINPUT messages. Each GM_HANDLEINPUT message corresponds to a single **InputEvent** structure. These **InputEvents** can be keyboard presses, timer events, mouse moves, or mouse button presses. The message's **gpi_IEvent** field points to this **InputEvent** structure. It's up to the GM_HANDLEINPUT method to interpret the meaning of these events and update the visual state of the gadget as the user manipulates the gadget. For example, the GM_HANDLEINPUT method of a prop gadget has to track mouse events to see where the user has moved the prop gadget's knob and update the gadget's imagery to reflect the new position of the knob.

For the GM_GOACTIVE method, the **gpi_IEvent** field points to the struct **InputEvent** that triggered the GM_GOACTIVE message. Unlike the GM_HANDLEINPUT message, GM_GOACTIVE's **gpi_IEvent** can be NULL. If the GM_GOACTIVE message was triggered by a function like intuition.library's **ActivateGadget()** and not by a real **InputEvent** (like the user clicking the gadget), the **gpi_IEvent** field will be NULL.

For gadgets that only want to become active as a direct result of a mouse click, this difference is important. For example, the prop gadget becomes active only when the user clicks on its knob. Because the only way the user can control the prop gadget is via the mouse, it does not make sense for anything but the mouse to activate the gadget. On the other hand, a string gadget doesn't care how it is activated because, as soon as it's active, it gets user input from the keyboard rather than the mouse. Not all gadgets can become active. Some gadgets cannot become active because they have been temporarily disabled (their **Gadget.Flags** GFLG_DISABLED bit is set). Other gadgets will not become active because they don't need to process input. For example, a toggle gadget won't become active because it only needs to process one input event, the mouse click that toggles the gadget (which it gets from the GM_GOACTIVE message). If a toggle gadget gets a GM_GOACTIVE message and its **gpi_IEvent** field is not NULL, it will toggle its state and refuse to "go active".

The GM_GOACTIVE method has to take care of any visual state changes to a gadget that a GM_GOACTIVE message might trigger. For example, the toggle gadget in the previous paragraph has to take care of toggling its visual state from selected imagery to unselected imagery. If the gadget goes through a state change when it becomes the active gadget, (like when a string gadget positions its cursor) GM_GOACTIVE has to take care of this.

The return values of both GM_GOACTIVE and GM_HANDLEINPUT tell Intuition whether or not the gadget wants to be active. A gadget's GM_GOACTIVE method returns GMR_MEACTIVE (defined in *<intuition/gadgetclass.h>*) if it wants to become the active gadget. A gadget's GM_HANDLEINPUT method returns GMR_MEACTIVE if it wants to remain the active gadget. If a gadget either does not want to become or remain the active gadget, it returns one of the "go inactive" return values:

GMR_NOREUSE	Tells Intuition to throw away the **gpInput.gpi_IEvent InputEvent**.
GMR_REUSE	Tells Intuition to process the **gpInput.gpi_IEvent InputEvent**.
GMR_NEXTACTIVE	Tells Intuition to throw away the **gpInput.gpi_IEvent InputEvent** and activate the next GFLG_TagCycle gadget.
GMR_PREVACTIVE	Tells Intuition to throw away the **gpInput.gpi_IEvent InputEvent** and activate the previous GFLG_TagCycle gadget.

GMR_NOREUSE tells Intuition that the gadget does not want to be active and to throw away the **InputEvent** that triggered the message. For example, an active prop gadget returns GMR_NOREUSE when the user lets go of the left mouse button (thus letting go of the prop gadget's knob).

For the GM_HANDLEINPUT method, a gadget can also return GMR_REUSE, which tells Intuition to reuse the **InputEvent**. For example, if the user clicks outside the active string gadget, that string gadget returns GMR_REUSE. Intuition can now process that mouse click, which can be over another gadget. Another case where a string gadget returns GMR_REUSE is when the user pushes the right mouse button (the menu button). The string gadget becomes inactive and the menu button **InputEvent** gets reused. Intuition sees this event and tries to pop up the menu bar.

For the GM_GOACTIVE method, a gadget must not return GMR_REUSE. If a gadget gets a GM_GOACTIVE message from Intuition and the message has an **gpi_IEvent**, the message was triggered by the user clicking on the gadget. In this case, Intuition knows that the user is trying to select the gadget. Intuition doesn't know if the gadget can be activated, but if it can be activated, the event that triggered the activation has just taken place. If the gadget cannot become active for any reason, it must not let Intuition reuse that **InputEvent** as the gadget has already taken care of the the event's purpose (clicking on the gadget). In essence, the user tried to activate the gadget and the gadget refused to become active.

The other two possible return values, GMR_NEXTACTIVE and GMR_PREVACTIVE were added to the OS for Release 2.04. These tell Intuition that a gadget does not want to be active and that the **InputEvent** should be discarded. Intuition then looks for the next (GMR_NEXTACTIVE) or previous (GMR_PREVACTIVE) gadget that has its GFLG_TABCYCLE flag set in its **Gadget.Activation** field (see the gadgetclass GA_TabCycle attribute in the *Boopsi Class Reference* in the Appendix B of this manual).

For both GM_GOACTIVE and GM_HANDLEINPUT, the gadget can bitwise-OR any of these ''go inactive'' return values with GMR_VERIFY. The GMR_VERIFY flag tells Intuition to send a GADGETUP **IntuiMessage** to the gadget's window. If the gadget uses GMR_VERIFY, it has to supply a value for the **IntuiMessage.Code** field. It does this by passing a value in the **gpInput.gpi_Termination** field. This field points to a long word, the lower 16-bits of which Intuition copies into the **Code** field. The upper 16-bits are for future enhancements, so clear these bits.

GM_GOINACTIVE

After an active gadget deactivates, Intuition sends it a GM_GOINACTIVE message (defined in *<intuition/gadgetclass.h>*):

```
struct gpGoInactive
{
    ULONG             MethodID;    /* GM_GOINACTIVE */
    struct GadgetInfo *gpgi_GInfo;

    /* V37 field only!  DO NOT attempt to read under V36! */
    ULONG             gpgi_Abort; /* gpgi_Abort=1 if gadget was aborted by Intuition   */
                                  /* and 0 if gadget went inactive at its own request. */
};
```

The **gpgi_Abort** field contains either a 0 or 1. If 0, the gadget became inactive on its own power (because the GM_GOACTIVE or GM_HANDLEINPUT method returned something besides GMR_MEACTIVE). If **gpgi_Abort** is 1, Intuition aborted this active gadget. Some instances where Intuition aborts a gadget include: the user clicked in another window or screen, an application removed the active gadget with **RemoveGList()**, and an application called **ActiveWindow()** on a window other than the gadget's window.

THE ACTIVE GADGET

While a gadget is active, Intuition sends it a GM_HANDLEINPUT message for every timer pulse, mouse move, mouse click, and key press that takes place. A timer event pulse arrives about every tenth of a second. Mouse move events can arrive at a much higher rate than the timer pulses. Without even considering the keyboard, a gadget can get a lot of GM_HANDLEINPUT messages in a short amount of time. Because the active gadget has to handle a large volume of GM_HANDLEINPUT messages, the overhead of this method should be kept to a minimum.

Because the gadget will always receive a GM_GOACTIVE message before it is active and a GM_GOINACTIVE message after it is no longer active, the gadget can use these methods to allocate, initialize, and deallocate temporary resources it needs for the GM_HANDLEINPUT method. This can significantly reduce the overhead of GM_HANDLEINPUT because it eliminates the need to allocate, initialize, and deallocate resources for every GM_HANDLEINPUT message.

Note that the **RastPort** from **ObtainGIRPort()** is not cachable using this method. If the GM_HANDLEINPUT method needs to use a **RastPort**, it has to obtain and release the **RastPort** for every GM_HANDLEINPUT message using **ObtainGIRPort()** and **ReleaseGIRPort()**.

RKMButtonclass.c

The following example is a sample Boopsi gadget, **RKMButClass.c**. While the user has the RKMButton selected, the gadget sends an OM_UPDATE message to its ICA_TARGET for every timer event the button sees. The gadget sends notification about its RKMBUT_Pulse attribute, which is the horizontal distance in screen pixels the mouse is from the center of the button. The gadget takes care of rendering all of its imagery (as opposed to using a Boopsi image to do it). The gadget's imagery is scalable to any dimensions and can be set (using **SetGadgetAttrs()**) while the gadget is in place.

One possible use for such a gadget is as buttons for a prop gadget. If the user has the prop gadget's RKMButton selected, while the mouse is to the left of the button's center, the knob on the prop gadget moves left. While the mouse is to the right of the button's center, the knob on the prop gadget moves right. The speed at which the knob moves is proportional to the horizontal distance from the mouse to the active RKMButton.

```
;/* RKMButClass.c - Example Boopsi gadget for RKRM:Libraries
; Execute me to compile me with Lattice 5.10b
LC -b1 -d0 -cfistq -v -y -j73 RKMButClass.c
Blink FROM LIB:c.o,RKMButClass.o TO TestBut LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/classes.h>
#include <intuition/classusr.h>
#include <intuition/imageclass.h>
#include <intuition/gadgetclass.h>
#include <intuition/cghooks.h>
#include <intuition/icclass.h>
#include <utility/tagitem.h>
#include <utility/hooks.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/utility_protos.h>
#include <clib/alib_protos.h>
```

```
#include <clib/alib_stdio_protos.h>

#include <graphics/gfxmacros.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\ $VER: TestBut 37.1";

/*************************************************************/
/***************       Class specifics      ***************/
/*************************************************************/
#define RKMBUT_Pulse    (TAG_USER + 1)

struct ButINST
{
    LONG midX, midY; /* Coordinates of middle of gadget */
};

/* ButINST has one flag:  */
#define ERASE_ONLY    0x00000001 /* Tells rendering routine to */
                                 /* only erase the gadget, not */
                                 /* rerender a new one.  This  */
                                 /* lets the gadget erase it-  */
                                 /* self before it rescales.   */

/* The functions in this module */
Class *initRKMButGadClass(void);
BOOL   freeRKMButGadClass(Class *);
ULONG  dispatchRKMButGad(Class *, Object *, Msg);
void   NotifyPulse(Class *, Object *, ULONG, LONG, struct gpInput *);
ULONG  RenderRKMBut(Class *, struct Gadget *, struct gpRender *);
void   geta4(void);
void   MainLoop(ULONG, ULONG);

/*************************************************************************************************/
/* The main() function connects an RKMButClass object to a Boopsi integer gadget, which displays */
/* the RKMButClass gadget's RKMBUT_Pulse value.  The code scales and move the gadget while it is */
/* in place.                                                                                     */
/*************************************************************************************************/

struct TagItem pulse2int[] =
{
    {RKMBUT_Pulse, STRINGA_LongVal},
    {TAG_END,}
};

#define INTWIDTH  40
#define INTHEIGHT 20

struct Library *IntuitionBase, *UtilityBase, *GfxBase;
struct Window *w;
Class *rkmbutcl;
struct Gadget *integer, *but;
struct IntuiMessage *msg;

void main(void)
{
    if (IntuitionBase = OpenLibrary("intuition.library", 37L))
    {
        if (UtilityBase = OpenLibrary("utility.library", 37L))
        {
            if (GfxBase = OpenLibrary("graphics.library", 37L))
            {
                if (w = OpenWindowTags(NULL,
                    WA_Flags,       WFLG_DEPTHGADGET | WFLG_DRAGBAR |
                                        WFLG_CLOSEGADGET | WFLG_SIZEGADGET,
                    WA_IDCMP,       IDCMP_CLOSEWINDOW,
                    WA_Width,           640,
                    WA_Height,          200,
                    TAG_END))
                {
                    WindowLimits(w, 450, 200, 640, 200);
```

```
                if (rkmbutcl = initRKMButGadClass())
                {
                    if (integer = (struct Gadget *)NewObject(NULL,
                                   "strgclass",
                                   GA_ID,             1L,
                                   GA_Top,            (w->BorderTop) + 5L,
                                   GA_Left,           (w->BorderLeft) + 5L,
                                   GA_Width,          INTWIDTH,
                                   GA_Height,         INTHEIGHT,
                                   STRINGA_LongVal,   0L,
                                   STRINGA_MaxChars,  5L,
                                   TAG_END))
                    {
                        if (but = (struct Gadget *)NewObject(rkmbutcl,
                                   NULL,
                                   GA_ID,        2L,
                                   GA_Top,       (w->BorderTop) + 5L,
                                   GA_Left,      integer->LeftEdge +
                                                     integer->Width + 5L,
                                   GA_Width,     40L,
                                   GA_Height,    INTHEIGHT,
                                   GA_Previous,  integer,
                                   ICA_MAP,      pulse2int,
                                   ICA_TARGET,   integer,
                                   TAG_END))
                        {
                            AddGList(w, integer, -1, -1, NULL);
                            RefreshGList(integer, w, NULL, -1);

                            SetWindowTitles(w,
                                "<-- Click to resize gadget Height",
                                NULL);
                            MainLoop(TAG_DONE, 0L);

                            SetWindowTitles(w,
                                "<-- Click to resize gadget Width",
                                NULL);
                            MainLoop(GA_Height, 100L);

                            SetWindowTitles(w,
                                "<-- Click to resize gadget Y position",
                                NULL);
                            MainLoop(GA_Width, 100L);

                            SetWindowTitles(w,
                                "<-- Click to resize gadget X position",
                                NULL);
                            MainLoop(GA_Top, but->TopEdge + 20);

                            SetWindowTitles(w,
                                "<-- Click to quit", NULL);
                            MainLoop(GA_Left, but->LeftEdge + 20);

                            RemoveGList(w, integer, -1);
                            DisposeObject(but);
                        }
                        DisposeObject(integer);
                    }
                    freeRKMButGadClass(rkmbutcl);
                    }
                CloseWindow(w);
            }
            CloseLibrary(GfxBase);
            }
            CloseLibrary(UtilityBase);
        }
        CloseLibrary(IntuitionBase);
    }
}

void MainLoop(ULONG attr, ULONG value)
{
    ULONG done = FALSE;

    SetGadgetAttrs(but, w, NULL, attr, value, TAG_DONE);
```

```
    while (done == FALSE)
    {
        WaitPort((struct MsgPort *)w->UserPort);
        while (msg = (struct IntuiMessage *)
           GetMsg((struct MsgPort *)w->UserPort))
        {
            if (msg->Class == IDCMP_CLOSEWINDOW)
            {
                done = TRUE;
            }
            ReplyMsg(msg);
        }
    }
}

/*************************************************************/
/**    Make the class and set up the dispatcher's hook     **/
/*************************************************************/
Class   *initRKMButGadClass(void)
{
    Class *cl = NULL;
    extern ULONG HookEntry();     /* defined in amiga.lib */

    if ( cl =  MakeClass( NULL,
                "gadgetclass", NULL,
                sizeof ( struct ButINST ),
                0 ))
    {
        /* initialize the cl_Dispatcher Hook    */
        cl->cl_Dispatcher.h_Entry = HookEntry;
        cl->cl_Dispatcher.h_SubEntry = dispatchRKMButGad;
    }
    return ( cl );
}

/*************************************************************/
/******************       Free the class      ****************/
/*************************************************************/
BOOL freeRKMButGadClass( Class *cl )
{
    return (FreeClass(cl));
}

/*************************************************************/
/**********        The RKMBut class dispatcher      *********/
/*************************************************************/
ULONG dispatchRKMButGad(Class *cl, Object *o, Msg msg)
{
    struct ButINST *inst;
    ULONG retval = FALSE;
    Object *object;

    /* SAS/C and Manx function to make sure register A4
       contains a pointer to global data */
    geta4();

    switch (msg->MethodID)
    {
        case OM_NEW:       /* First, pass up to superclass */
            if (object = (Object *)DoSuperMethodA(cl, o, msg))
            {
                struct Gadget *g = (struct Gadget *)object;

                            /* Initial local instance data */
                inst = INST_DATA(cl, object);
                inst->midX   = g->LeftEdge + ( (g->Width) / 2);
                inst->midY   = g->TopEdge + ( (g->Height) / 2);

                retval = (ULONG)object;
            }
            break;
        case GM_HITTEST:
                   /* Since this is a rectangular gadget this  */
                   /* method always returns GMR_GADGETHIT.     */
            retval = GMR_GADGETHIT;
            break;
```

```
case GM_GOACTIVE:
    inst = INST_DATA(cl, o);

            /* Only become active if the GM_GOACTIVE   */
            /* was triggered by direct user input.     */
    if (((struct gpInput *)msg)->gpi_IEvent)
    {
               /* This gadget is now active, change    */
               /* visual state to selected and render. */
        ((struct Gadget *)o)->Flags |= GFLG_SELECTED;
                        RenderRKMBut(cl, (struct Gadget *)o, (struct gpRender *)msg);
        retval = GMR_MEACTIVE;
    }
    else            /* The GM_GOACTIVE was not         */
                    /* triggered by direct user input. */
        retval = GMR_NOREUSE;
    break;
case GM_RENDER:
    retval = RenderRKMBut(cl, (struct Gadget *)o, (struct gpRender *)msg);
    break;
case GM_HANDLEINPUT:   /* While it is active, this gadget sends its superclass an        */
                       /* OM_NOTIFY pulse for every IECLASS_TIMER event that goes by     */
                       /* (about one every 10th of a second).  Any object that is        */
                       /* connected to this gadget will get A LOT of OM_UPDATE messages. */
    {
        struct Gadget *g = (struct Gadget *)o;
        struct gpInput *gpi = (struct gpInput *)msg;
        struct InputEvent *ie = gpi->gpi_IEvent;

        inst = INST_DATA(cl, o);

        retval = GMR_MEACTIVE;

        if (ie->ie_Class == IECLASS_RAWMOUSE)
        {
            switch (ie->ie_Code)
            {
                case SELECTUP: /* The user let go of the gadget so return GMR_NOREUSE    */
                               /* to deactivate and to tell Intuition not to reuse       */
                               /* this Input Event as we have already processed it.      */

                               /*If the user let go of the gadget while the mouse was    */
                               /*over it, mask GMR_VERIFY into the return value so       */
                               /*Intuition will send a Release Verify (GADGETUP).        */
                    if ( ((gpi->gpi_Mouse).X < g->LeftEdge) ||
                         ((gpi->gpi_Mouse).X > g->LeftEdge + g->Width) ||
                         ((gpi->gpi_Mouse).Y < g->TopEdge) ||
                         ((gpi->gpi_Mouse).Y > g->TopEdge + g->Height) )
                        retval = GMR_NOREUSE | GMR_VERIFY;
                    else
                        retval = GMR_NOREUSE;

                                   /* Since the gadget is going inactive, send a final   */
                                   /* notification to the ICA_TARGET.                    */
                    NotifyPulse(cl , o, 0L, inst->midX, (struct gp_Input *)msg);
                    break;
                case MENUDOWN: /* The user hit the menu button. Go inactive and let      */
                               /* Intuition reuse the menu button event so Intuition can */
                               /* pop up the menu bar.                                   */
                    retval = GMR_REUSE;

                                   /* Since the gadget is going inactive, send a final   */
                                   /* notification to the ICA_TARGET.                    */
                    NotifyPulse(cl , o, 0L, inst->midX, (struct gp_Input *)msg);
                    break;
                default:
                    retval = GMR_MEACTIVE;
            }

        }
        else if (ie->ie_Class == IECLASS_TIMER)
                      /* If the gadget gets a timer event, it sends an interim OM_NOTIFY */
            NotifyPulse(cl, o, OPUF_INTERIM, inst->midX, gpi); /*     to its superclass. */
    }
    break;
```

```
    case GM_GOINACTIVE:           /* Intuition said to go inactive.  Clear the GFLG_SELECTED */
                                  /* bit and render using unselected imagery.                */
        ((struct Gadget *)o)->Flags &= ~GFLG_SELECTED;
        RenderRKMBut(cl, (struct Gadget *)o, (struct gpRender *)msg);
        break;
    case OM_SET:/* Although this class doesn't have settable attributes, this gadget class   */
                /* does have scaleable imagery, so it needs to find out when its size and/or */
                /* position has changed so it can erase itself, THEN scale, and rerender.    */
        if ( FindTagItem(GA_Width,  ((struct opSet *)msg)->ops_AttrList) ||
             FindTagItem(GA_Height, ((struct opSet *)msg)->ops_AttrList) ||
             FindTagItem(GA_Top,    ((struct opSet *)msg)->ops_AttrList) ||
             FindTagItem(GA_Left,   ((struct opSet *)msg)->ops_AttrList) )
        {
            struct RastPort *rp;
            struct Gadget *g = (struct Gadget *)o;

            WORD x,y,w,h;

            x = g->LeftEdge;
            y = g->TopEdge;
            w = g->Width;
            h = g->Height;

            inst = INST_DATA(cl, o);

            retval = DoSuperMethodA(cl, o, msg);

                                                  /* Get pointer to RastPort for gadget. */
            if (rp = ObtainGIRPort( ((struct opSet *)msg)->ops_GInfo) )
            {
                        UWORD *pens = ((struct opSet *)msg)->ops_GInfo->gi_DrInfo->dri_Pens;

                SetAPen(rp, pens[BACKGROUNDPEN]);
                SetDrMd(rp, JAM1);                          /* Erase the old gadget.       */
                RectFill(rp, x, y, x+w, y+h);

                inst->midX = g->LeftEdge + ( (g->Width) / 2); /* Recalculate where the       */
                inst->midY = g->TopEdge + ( (g->Height) / 2); /* center of the gadget is.    */

                                                            /* Rerender the gadget.        */
                DoMethod(o, GM_RENDER, ((struct opSet *)msg)->ops_GInfo, rp, GREDRAW_REDRAW);
                ReleaseGIRPort(rp);
            }
        }
        else
            retval = DoSuperMethodA(cl, o, msg);
        break;
    default:            /* rkmmodelclass does not recognize the methodID, let the superclass's */
                        /* dispatcher take a look at it.                                       */
        retval = DoSuperMethodA(cl, o, msg);
        break;
    }
    return(retval);
}


/*************************************************************************************************/
/************** Build an OM_NOTIFY message for RKMBUT_Pulse and send it to the superclass. *******/
/*************************************************************************************************/
void NotifyPulse(Class *cl, Object *o, ULONG flags, LONG mid, struct gpInput *gpi)
{
    struct TagItem tt[3];

    tt[0].ti_Tag = RKMBUT_Pulse;
    tt[0].ti_Data = mid - ((gpi->gpi_Mouse).X + ((struct Gadget *)o)->LeftEdge);

    tt[1].ti_Tag = GA_ID;
    tt[1].ti_Data = ((struct Gadget *)o)->GadgetID;

    tt[2].ti_Tag = TAG_DONE;

    DoSuperMethod(cl, o, OM_NOTIFY, tt, gpi->gpi_GInfo, flags);
}
```

```
/***************************************************************************************************/
/******************************   Erase and rerender the gadget.   ******************************/
/***************************************************************************************************/
ULONG RenderRKMBut(Class *cl, struct Gadget *g, struct gpRender *msg)
{
    struct ButINST *inst = INST_DATA(cl, (Object *)g);
    struct RastPort *rp;
    ULONG retval = TRUE;
    UWORD *pens = msg->gpr_GInfo->gi_DrInfo->dri_Pens;

    if (msg->MethodID == GM_RENDER)   /* If msg is truly a GM_RENDER message (not a gpInput that */
                                      /* looks like a gpRender), use the rastport within it...   */
        rp = msg->gpr_RPort;
    else                              /* ...Otherwise, get a rastport using ObtainGIRPort().     */
        rp = ObtainGIRPort(msg->gpr_GInfo);

    if (rp)
    {
        UWORD back, shine, shadow, w, h, x, y;

        if (g->Flags & GFLG_SELECTED) /* If the gadget is selected, reverse the meanings of the  */
        {                             /* pens.                                                    */
            back   = pens[FILLPEN];
            shine  = pens[SHADOWPEN];
            shadow = pens[SHINEPEN];
        }
        else
        {
            back   = pens[BACKGROUNDPEN];
            shine  = pens[SHINEPEN];
            shadow = pens[SHADOWPEN];
        }
        SetDrMd(rp, JAM1);

        SetAPen(rp, back);          /* Erase the old gadget.       */
        RectFill(rp, g->LeftEdge,
                     g->TopEdge,
                     g->LeftEdge + g->Width,
                     g->TopEdge + g->Height);

        SetAPen(rp, shadow);     /* Draw shadow edge.            */
        Move(rp, g->LeftEdge + 1, g->TopEdge + g->Height);
        Draw(rp, g->LeftEdge + g->Width, g->TopEdge + g->Height);
        Draw(rp, g->LeftEdge + g->Width, g->TopEdge + 1);

        w = g->Width / 4;       /* Draw Arrows - Sorry, no frills imagery */
        h = g->Height / 2;
        x = g->LeftEdge + (w/2);
        y = g->TopEdge + (h/2);

        Move(rp, x, inst->midY);
        Draw(rp, x + w, y);
        Draw(rp, x + w, y + (g->Height) - h);
        Draw(rp, x, inst->midY);

        x = g->LeftEdge + (w/2) + g->Width / 2;

        Move(rp, x + w, inst->midY);
        Draw(rp, x, y);
        Draw(rp, x, y  + (g->Height) - h);
        Draw(rp, x + w, inst->midY);

        SetAPen(rp, shine);    /* Draw shine edge.           */
        Move(rp, g->LeftEdge, g->TopEdge + g->Height - 1);
        Draw(rp, g->LeftEdge, g->TopEdge);
        Draw(rp, g->LeftEdge + g->Width - 1, g->TopEdge);

        if (msg->MethodID != GM_RENDER) /* If we allocated a rastport, give it back.            */
            ReleaseGIRPort(rp);
    }
    else retval = FALSE;
    return(retval);
}
```

Function Reference

The following are brief descriptions of the Intuition and *amiga.lib* functions discussed in this chapter. See the "Amiga ROM Kernel Reference Manual: Includes and Autodocs" for details on each function call. All these functions require Release 2 or a later version of the Amiga operating system.

Table 12-1: Intuition Library Boopsi Functions

Function	Description
NewObjectA()	Create a new Boopsi object (tag array form).
NewObject()	Create a new Boopsi object (varargs form).
DisposeObject()	Dispose of a Boopsi object.
SetAttrs()	Set one or more of a Boopsi object's attributes (tag array form).
SetGadgetAttrs()	Set one or more of a Boopsi object's attributes (varargs form).
GetAttr()	Obtain an attribute from a Boopsi object.
MakeClass()	Create a new private or public Boopsi class.
FreeClass()	Free a Boopsi class created by **MakeClass()**.
AddClass()	Add a public Boopsi class to Intuition's internal list of public classes.
RemoveClass()	Remove a public Boopsi class that was added to Intuition's internal list with **AddClass()**.
ObtainGIRPort()	Set up a **RastPort** for use by a Boopsi gadget dispatcher.
ReleaseGIRPort()	Free a **RastPort** set up by **ReleaseGIRPort()**.

Table 12-2: Amiga.lib Boopsi Functions

Function	Description
DoMethodA()	Send a Boopsi message to a Boopsi object (tag array form).
DoMethod()	Send a Boopsi message to a Boopsi object (varargs form).
DoSuperMethodA()	Send a Boopsi message to a Boopsi object as if the object was an instance of its class's superclass (tag array form).
DoSuperMethod()	Send a Boopsi message to a Boopsi object as if the object was an instance of its class's superclass (varargs form).
CoerceMethodA()	Send a Boopsi message to a Boopsi object as if the object was an instance of the specified class (tag array form).
CoerceMethod()	Send a Boopsi message to a Boopsi object as if the object was an instance of the specified class (varargs form).
SetSuperAttrs()	Send a Boopsi OM_SET message to the Boopsi object's superclass.

Chapter 13
PREFERENCES

To make the Amiga operating system easily configurable by the user, the OS comes with a family of editors and associated data files known collectively as Preferences. Preferences allows the user to set system-wide configuration options such as the printer driver to use, serial port baud rate and other items. To make an application appealing to the user, the system-wide Preferences settings should be respected as much as possible by applications. This chapter describes how to use the Preferences system in your programs.

In Release 2 the number of Preference items and the way they are handled is very different from 1.3 and earlier versions, though there is backward compatibility with old Preferences items. This chapter describes both the old 1.3 and the new Release 2 Preferences.

Preferences in 1.3 and Older Versions of the OS

In 1.3, the Preferences program allows the user to see and change many system wide parameters, like the Workbench colors, pointer image, printer settings etc. When a Preferences item is changed, the new setting can be used temporarily (until a reset occurs) or stored permanently in the devs:system-configuration file. Under 1.3, all Preferences items are stored in this file which the system reads at boot time to find out how to set various system-wide options.

The 1.3 Preferences system allows the user to change the following items:

- Date and time of day. These are automatically saved in the battery-backed clock, if one is present.
- Key repeat speed - the speed at which a key repeats when held down.
- Key repeat delay - the amount of delay before a key begins repeating.
- Mouse speed - how fast the pointer moves when the user moves the mouse.
- Double-click delay - maximum time allowed between the two clicks of a mouse double click. For information about how to test for double-click timeout, see the description of the **DoubleClick()** function in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

- Text size - size of the default font characters. The user can choose 64-column mode (64 characters on a line in high-resolution and 32 characters in low-resolution mode) or 80 column mode (80 characters on a line in high-resolution and 40 characters in low-resolution mode). The first variable in the **Preferences** structure is **FontHeight**, which is the height of the characters in display lines. If this is equal to the constant TOPAZ_EIGHTY, the user has chosen the 80-column mode. If it is equal to TOPAZ_SIXTY, the user has chosen the 64-column mode. Note that certain utility programs allow the user to change the default font under 1.3, so you cannot rely on the default font being Topaz 8 or 9.

- Display centering - allows the user to center the image on the video display.

- Serial port - the user can change the baud rate and other serial port parameters to accommodate whatever device is attached to the serial connector. Normally you use these values as defaults when you open the serial device. If you change the baud rate or other serial port options locally, it is good practice to reset them to the values specified in Preferences before quitting.

- Workbench colors - the user can change any of the four colors in the 1.3 Workbench screen by adjusting the amounts of red, green and blue in each color.

- Printer - the user can select from a number of printers supported by the Amiga and also indicate whether the printer is connected to the serial connector or the parallel connector.

- Print characteristics - the user can select paper size, right and left margin, continuous feed or single sheets, draft or letter quality, pitch and line spacing. For graphic printing, the user can specify the density, scaling method, select a vertical or horizontal dump, etc.

READING 1.3 PREFERENCES

Applications can obtain a copy of Preferences by calling the Intuition function **GetPrefs()**. In a system in which there is no devs:system-configuration file, **GetDefPrefs()** can be used to obtain the Intuition default Preference settings.

```
struct Preferences *GetPrefs(struct Preferences *preferences, LONG size);
struct Preferences *GetDefPrefs(struct Preferences *preferences, LONG size);
```

GetPrefs() and **GetDefPrefs()** have two arguments, a pointer to a buffer to receive the copy of the user Preferences and the size of this buffer. The most commonly used data is grouped near the beginning of the **Preferences** structure and you are free to read only as much as you need. So, if you are only interested in the first part of the **Preferences** structure, you do not need to allocate a buffer large enough to hold the entire structure. These functions return a pointer to your buffer if successful, NULL otherwise.

If you are using Intuition IDCMP for input, you can set the IDCMP flag IDCMP_NEWPREFS (formerly the NEWPREFS flag under V34 and earlier versions of the OS). With this flag set, your program will receive an **IntuiMessage** informing you changes have been made to Preferences. To get the latest settings, you would again call **GetPrefs()**.

PREFERENCES STRUCTURE IN 1.3

The **Preferences** structure in 1.3 and earlier versions of the OS is a static 232 byte data structure defined in *<intuition/preferences.h>* as follows:

```
struct Preferences
{
    /* the default font height */
    BYTE FontHeight;                    /* height for system default font  */

    /* constant describing what's hooked up to the port */
    UBYTE PrinterPort;                  /* printer port connection         */

    /* the baud rate of the port */
    UWORD BaudRate;                     /* baud rate for the serial port   */

    /* various timing rates */
    struct timeval KeyRptSpeed;         /* repeat speed for keyboard       */
    struct timeval KeyRptDelay;         /* Delay before keys repeat        */
    struct timeval DoubleClick;         /* Interval allowed between clicks */

    /* Intuition Pointer data */
    UWORD PointerMatrix[POINTERSIZE];   /* Definition of pointer sprite    */
    BYTE XOffset;                       /* X-Offset for active 'bit'       */
    BYTE YOffset;                       /* Y-Offset for active 'bit'       */
    UWORD color17;                      /***********************************/
    UWORD color18;                      /* Colours for sprite pointer      */
    UWORD color19;                      /***********************************/
    UWORD PointerTicks;                 /* Sensitivity of the pointer      */

    /* Workbench Screen colors */
    UWORD color0;                       /***********************************/
    UWORD color1;                       /*  Standard default colours       */
    UWORD color2;                       /*   Used in the Workbench         */
    UWORD color3;                       /***********************************/

    /* positioning data for the Intuition View */
    BYTE ViewXOffset;                   /* Offset for top lefthand corner  */
    BYTE ViewYOffset;                   /* X and Y dimensions              */
    WORD ViewInitX, ViewInitY;          /* View initial offset values      */

    BOOL EnableCLI;                     /* CLI availability switch (OBSOLETE)*/

    /* printer configurations */
    UWORD PrinterType;                  /* printer type                    */
    UBYTE PrinterFilename[FILENAME_SIZE]; /* file for printer              */

    /* print format and quality configurations */
    UWORD PrintPitch;                   /* print pitch                     */
    UWORD PrintQuality;                 /* print quality                   */
    UWORD PrintSpacing;                 /* number of lines per inch        */
    UWORD PrintLeftMargin;              /* left margin in characters       */
    UWORD PrintRightMargin;             /* right margin in characters      */
    UWORD PrintImage;                   /* positive or negative            */
    UWORD PrintAspect;                  /* horizontal or vertical          */
    UWORD PrintShade;                   /* b&w, half-tone, or color        */
    WORD PrintThreshold;                /* darkness ctrl for b/w dumps     */

    /* print paper descriptors */
    UWORD PaperSize;                    /* paper size                      */
    UWORD PaperLength;                  /* paper length in number of lines */
    UWORD PaperType;                    /* continuous or single sheet      */

    /* Serial device settings: These are six nibble-fields in three bytes */
    /* (these look a little strange so the defaults will map out to zero) */
    UBYTE   SerRWBits;    /* upper nibble = (8-number of read bits)        */
                          /* lower nibble = (8-number of write bits)       */
    UBYTE   SerStopBuf;   /* upper nibble = (number of stop bits - 1)      */
                          /* lower nibble = (table value for BufSize)      */
    UBYTE   SerParShk;    /* upper nibble = (value for Parity setting)     */
                          /* lower nibble = (value for Handshake mode)     */
    UBYTE   LaceWB;       /* if workbench is to be interlaced              */
```

```
    UBYTE    WorkName[FILENAME_SIZE]; /* temp file for printer          */

    BYTE     RowSizeChange;          /* affect NormalDisplayRows/Columns  */
    BYTE     ColumnSizeChange;

    UWORD     PrintFlags;        /* user preference flags                  */
    UWORD     PrintMaxWidth;     /* max width of printed picture in 10ths/in  */
    UWORD     PrintMaxHeight;    /* max height of printed picture in 10ths/in */
    UBYTE     PrintDensity;      /* print density                          */
    UBYTE     PrintXOffset;      /* offset of printed picture in 10ths/inch   */

    UWORD     wb_Width;          /* override default workbench width  */
    UWORD     wb_Height;         /* override default workbench height */
    UBYTE     wb_Depth;          /* override default workbench depth  */

    UBYTE     ext_size;          /* extension information -- do not touch! */
                                 /* extension size in blocks of 64 bytes   */
};
```

SETTING 1.3 PREFERENCES

The instance of the **Preferences** structure in memory can be changed with the Intuition **SetPrefs()** function:

```
struct Preferences *SetPrefs(struct Preferences *preferences, LONG size, BOOL inform);
```

In addition to a buffer holding the **Preferences** structure, and the buffer size, this function takes an argument which indicates whether an IDCMP_NEWPREFS message should be broadcast to windows which have this flag set in the **Window.IDCMPFlags** field of their window.

> *Avoid Using SetPrefs().* This function is normally only used by Preferences-like utilities. There should be no need for a normal application to set the system Preferences with **SetPrefs()**.

ALTERNATIVES TO SETPREFS

Since the Amiga is a multitasking system, it is rarely correct for a single Amiga application to modify the user's system-wide Preferences. Instead, use methods such as the following to modify only your own application's appearance or behavior.

- Custom screen applications can control their own display mode, resolution, palette, and fonts. Use functions such a **LoadRGB4()** to change your own screen's palette, and **SetFont()** to change your own screen and window fonts. Workbench applications should never change the attributes of the user's Workbench.
- The mouse pointer for a window may be changed with **SetPointer()**.
- Serial device settings can be changed with the command SDCMD_SETPARAMS.
- Printer device settings may be changed by altering the printer's copy of the Preferences structure when you have the printer open. Note that Amiga applications should only keep the printer open while they are printing. This allows other applications to print, and also allows user changes to Printer Preferences to take effect.

See the Inution and graphics chapters of this manual, and the "Printer Device" and "Serial Device" chapters of the *Amiga ROM Kernel Reference Manual: Devices* for more information.

Preferences in Release 2

Under Release 2 (V36), the way Preferences are handled is significantly different. No longer is there one Preferences program with one configuration file. Instead there can be any number of Preferences editors (there are currently 13), each with its own separate configuration file covering a specific area. All these Preferences editors have the same look and feel. Using separate Preferences editors and configuration files allows for adding new Preferences items (and editors) in future versions of the OS.

PREFERENCES EDITORS AND STORAGE

In Release 2, the devs:system-configuration file has been replaced by various .prefs files, located in the ENV:sys and ENVARC:sys directories. System Preferences options currently in use are located in ENV:sys. Permanent, saved copies of system Preferences files are stored in ENVARC:sys. The contents of ENVARC: is copied at boot time to ENV:. Applications may also store their own preference files in ENV: but should use a subdirectory for that purpose.

Currently the following Preferences editors and files are available:

Table 13-1: Preferences Editors in Release 2

Preferences Editor	Preferences Configuration File
IControl	icontrol.prefs
Input	input.prefs
Palette	palette.ilbm
Pointer	pointer.ilbm
Printer	printer.prefs
PrinterGfx	printergfx.prefs
Overscan	overscan.prefs
ScreenMode	screenmode.prefs
Serial	serial.prefs
---	wbconfig.prefs
Font	wbfont.prefs, sysfont.prefs and screenfont.prefs
Time	---
WBPattern	wb.pat and win.pat

Each .prefs file is managed by editor with the same name, except for wbconfig.prefs, which is written directly by Workbench and has no editor. One Preferences editor has no .prefs file, Time. That Preferences editor writes directly to the battery backed clock.

When the user makes a change to a Preferences item with one of the editors, the changes will be saved in either ENV:sys or *both* ENV:sys and ENVARC:sys depending on whether the user saves the changes with the "Use" gadget or "Save" gadget of the Preferences editor.

The "Use" gadget is for making temporary changes and the new preferences will be stored only in ENV:sys. If the user reboots, the old preferences will be restored from the permanent copy in ENVARC:sys.

The "Save" gadget is for making permanent changes and the new preferences will be stored in both ENV:sys and ENVARC:sys. That way, if the user reboots, the new preferences will still be in effect since the system looks in ENVARC:sys to find out what preferences should be set to at boot time.

THE ENV: DIRECTORY AND NOTIFICATION

One advantage of the new Preferences system in Release 2 is file notification. File notification is a form of interprocess communication available in Release 2 that allows an application to be automatically notified if a change is made to a specific file or directory. This makes it easy for the application to react to changes the user makes to Preferences files.

File notification is also used by the system itself. The Release 2 Preferences control program, IPrefs, sets up notification on most of the Preferences files in ENV:sys. If the user alters a Preferences item (normally this is done with a Preferences editor), the system will notify IPrefs about the change and IPrefs will attempt to alter the user's environment to reflect the change.

For example, if the user opens the ScreenMode Preferences editor and changes the Workbench screen mode to high-resolution, the new settings are saved in Screenmode.prefs in the ENV:sys directory. IPrefs sets up notification on this file at boot time, so the file system will notify IPrefs of the change. IPrefs will read in the Screenmode.prefs file and reset the Workbench screen to high resolution mode.

Here's a short example showing how to set up notification on the serial.prefs file in ENV:sys. The program displays a message in a window whenever this file is changed (e.g., when the user selects the ''Use'' or ''Save'' gadget in the Serial Preferences editor).

```
;/* prefnotify.c.  - Execute me to compile me with SAS/C 5.10
lc -cfistq -v -y -j73 prefnotify.c
blink from LIB:c.o,prefnotify.o to prefnotify lib LIB:LC.lib LIB:amiga.lib
quit

** prefnotify.c - notified if serial prefs change
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <dos/notify.h>

#include <stdio.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

#ifdef LATTICE
int CXBRK(void)  { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define PREFSFILENAME "ENV:sys/serial.prefs"

static UBYTE   *VersTag = "\0$VER: prefnot 37.1 (09.07.91)";

extern struct Library *DOSBase;


void main(int argc, char **argv)
{
BOOL done=FALSE;
struct NotifyRequest *notifyrequest;
UBYTE          *filename;
LONG           signum;
ULONG          signals;

/* We need at least V37 for notification */
if (DOSBase->lib_Version >= 37)
    {
    /* Allocate a NotifyRequest structure */
    if (notifyrequest = AllocMem(sizeof(struct NotifyRequest), MEMF_CLEAR))
        {
        /* And allocate a signalsbit */
```

```
        if ((signum = AllocSignal(-1L)) != -1)
            {
            /* Initialize notification request */
            filename = PREFSFILENAME;
            notifyrequest->nr_Name = filename;
            notifyrequest->nr_Flags = NRF_SEND_SIGNAL;
            /* Signal this task */
            notifyrequest->nr_stuff.nr_Signal.nr_Task = (struct Task *) FindTask(NULL);
            /* with this signals bit */
            notifyrequest->nr_stuff.nr_Signal.nr_SignalNum = signum;

            if ((StartNotify(notifyrequest)) == DOSTRUE)
                {
                printf("Select Serial Prefs SAVE or USE to notify this program\n");
                printf("CTRL-C to exit\n\n");
                /* Loop until Ctrl-C to exit */
                while(!done)
                    {
                    signals = Wait(  (1L << signum) | SIGBREAKF_CTRL_C  );
                    if (signals & (1L << signum))
                        printf("Notification signal received.\n");
                    if (signals & SIGBREAKF_CTRL_C)
                        {
                        EndNotify(notifyrequest);
                        done=TRUE;
                        }
                    }
                }
            else printf("Can't start notification\n");
            FreeSignal(signum);
            }
        else printf("No signals available\n");
        FreeMem(notifyrequest, sizeof(struct NotifyRequest));
        }
    else printf("Not enough memory for NotifyRequest.\n");
    }
else printf("Requires at least V37 dos.library\n");
}
```

PREFERENCE FILE FORMAT IN RELEASE 2

To understand the format of Preferences files, you must be familiar with IFF file standard (see the *Amiga ROM Kernel Reference Manual: Devices* for the complete specification).

In general all Preferences files are stored in the IFF format with a type of PREF (see the exceptions noted below). Each file contains at least two **Chunks**, a header **Chunk** and a data **Chunk**.

The Header Chunk

The PRHD header chunk, contains a **PrefHeader** structure:

```
struct PrefHeader
{
    UBYTE ph_Version;
    UBYTE ph_Type;
    ULONG ph_Flags;
};
```

Currently all the fields are set to NULL. In future revisions these fields may be used to indicate a particular version and contents of a PREF chunk.

The Data Chunk

The data **Chunk** that follows the header **Chunk** depends on the kind of Preferences data the file contains. The types of Preferences data **Chunks** that are currently part of the system are:

Table 13-2: IFF Chunk Types in Release 2 Preferences Data Files

Chunk Name	Used With
FONT	Fonts, used for all font Preferences files. In future the **PrefHeader** may indicate what the font is used for.
ICTL	IControl
INPT	Input
OSCN	Overscan
PGFX	PrinterGfx
PTXT	PrinterText
SCRM	ScreenMode
SERL	Serial

Each chunk contains a structure applicable to the type.

FONT

```
struct FontPrefs
{
    LONG             fp_Reserved[4];
    UBYTE            fp_FrontPen;            /* Textcolor */
    UBYTE            fp_BackPen;             /* Character background color */
    UBYTE            fp_DrawMode;
    struct TextAttr  fp_TextAttr;
    BYTE             fp_Name[FONTNAMESIZE];  /* Font name */
};
```

ICTL

```
struct IControlPrefs
{
    LONG  ic_Reserved[4];       /* System reserved             */
    UWORD ic_TimeOut;           /* Verify timeout              */
    WORD  ic_MetaDrag;          /* Meta drag mouse event       */
    ULONG ic_Flags;             /* IControl flags (see below)  */
    UBYTE ic_WBtoFront;         /* CKey: WB to front           */
    UBYTE ic_FrontToBack;       /* CKey: front screen to back  */
    UBYTE ic_ReqTrue;           /* CKey: Requester TRUE        */
    UBYTE ic_ReqFalse;          /* CKey: Requester FALSE       */
};
```

The **ic_Flags** field can have the following values:

ICF_COERCE_COLORS
: This indicates that a displaymode with a matching number of colors has preference over a correct aspect ration when screen coercing takes place.

ICF_COERCE_LACE
: This indicates that chosing an interlaced display mode is allowed when coercing screens. Otherwise a non-interlaced display mode will be selected.

ICF_STRGAD_FILTER
: This indicates that control characters should be filtered out of string gadget user input.

ICF_MENUSNAP
: This indicates that an autoscroll screen should be snapped back to origin when the mouse menu-button is selected.

Note that the command key values in the last four fields of the **IControlPrefs** structure are ANSI codes, not RAWKEY codes.

INPT

```
struct InputPrefs
{
    LONG           ip_Reserved[4];
    UWORD          ip_PointerTicks; /* Sensitivity of the pointer */
    struct timeval ip_DoubleClick;  /* Interval between clicks */
    struct timeval ip_KeyRptDelay;  /* keyboard repeat delay   */
    struct timeval ip_KeyRptSpeed;  /* Keyboard repeat speed   */
    WORD           ip_MouseAccel;   /* Mouse acceleration      */
};
```

OSCN

```
struct OverscanPrefs
{
    ULONG            os_Reserved[4];
    ULONG            os_DisplayID;     /* Displaymode ID */
    Point            os_ViewPos;       /* View X/Y Offset */
    Point            os_Text;          /* TEXT overscan dimension */
    struct Rectangle os_Standard;      /* STANDARD overscan dimension */
};
```

PGFX

```
struct PrinterGfxPrefs
{
    LONG  pg_Reserved[4];
    UWORD pg_Aspect;                  /* Horizontal or vertical */
    UWORD pg_Shade;                   /* B&W, Greyscale, Color */
    UWORD pg_Image;                   /* Positive or negative image */
    WORD  pg_Threshold;               /* Black threshold */
    UBYTE pg_ColorCorrect;            /* RGB color correction */
    UBYTE pg_Dimensions;              /* Dimension type */
    UBYTE pg_Dithering;               /* Type of dithering */
    UWORD pg_GraphicFlags;            /* Rastport dump flags */
    UBYTE pg_PrintDensity;            /* Print density 1 - 7 */
    UWORD pg_PrintMaxWidth;           /* Maximum width */
    UWORD pg_PrintMaxHeight;          /* Maximum height */
    UBYTE pg_PrintXOffset;            /* X Offset */
    UBYTE pg_PrintYOffset;            /* Y Offset */
};
```

The possible values of each field are defined in *<prefs/printergfx.h>*. Note that your application is responsible for checking if the supplied values are valid.

PTXT

```
struct PrinterTxtPrefs
{
    LONG  pt_Reserved[4];                  /* System reserved           */
    UBYTE pt_Driver[DRIVERNAMESIZE];       /* printer driver filename   */
    UBYTE pt_Port;                         /* printer port connection   */

    UWORD pt_PaperType;                    /* Fanfold or single */
    UWORD pt_PaperSize;                    /* Standard, Legal, A4, A3 etc. */
    UWORD pt_PaperLength;                  /* Paper length in # of lines */

    UWORD pt_Pitch;                        /* Pica or Elite */
    UWORD pt_Spacing;                      /* 6 or 8 LPI */
    UWORD pt_LeftMargin;                   /* Left margin */
    UWORD pt_RightMargin;                  /* Right margin */
    UWORD pt_Quality;                      /* Draft or Letter */
};
```

SCRM

```
struct ScreenModePrefs
{
    ULONG sm_Reserved[4];
    ULONG sm_DisplayID;                      /* Displaymode ID */
    UWORD sm_Width;                          /* Screen width */
    UWORD sm_Height;                         /* Screen height */
    UWORD sm_Depth;                          /* Screen depth */
    UWORD sm_Control;                        /* BIT 0, Autoscroll yes/no */
};
```

SERL

```
struct SerialPrefs
{
    LONG  sp_Reserved[4];                  /* System reserved                */
    ULONG sp_BaudRate;                     /* Baud rate                      */

    ULONG sp_InputBuffer;                  /* Input buffer: 0 - 64K        */
    ULONG sp_OutputBuffer;                 /* Future: Output: 0 - 64K, def 0 */

    UBYTE sp_InputHandshake;               /* Input handshaking              */
    UBYTE sp_OutputHandshake;              /* Future: Output handshaking     */

    UBYTE sp_Parity;                       /* Parity                         */
    UBYTE sp_BitsPerChar;                  /* I/O bits per character         */
    UBYTE sp_StopBits;                     /* Stop bits                      */
};
```

OTHER PREFERENCES FILE FORMATS IN RELEASE 2

Not every Preferences file is stored as an IFF file of type PREF. The palette.ilbm and pointer.ilbm files contain a regular ILBM FORM to store their imagery. The win.pat and wb.pat files use a raw format with 16 bytes reserved, followed by a WORD giving the total size of the pattern, a WORD giving the bitplane count, and byte arrays (currently 32 bytes) for each bitplane. The format of the wbconfig.prefs file is private.

READING A PREFERENCES FILE

The following example shows a way to read a Preferences file.

```
;/* showprefs.c - Execute me to compile me with SAS C 5.10
LC -b0 -d0 -cfis -v -j73 showprefs.c
Blink FROM showprefs.o TO showprefs LIBRARY LIB:Amiga.lib
quit

** showprefs.c - parse and show some info from an IFF Preferences file
** NOTE: This example requires upcoming 2.1 prefs/ include files.
**
** IMPORTANT!! This example is not linked with startup code (eg. c.o).
** It uses strictly direct AmigaDOS stdio, and also demonstrates
** direct ReadArgs argument parsing.  Therefore it is a CLI-only
** example.  If launched from Workbench, packet errors would occur
** since the WbStartup message is still sitting in the process's
** pr_MsgPort, and the code would never be unloaded from memory.
*/

#include <exec/types.h>
#include <dos/dos.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>
#include <prefs/prefhdr.h>
#include <prefs/font.h>
#include <prefs/icontrol.h>
#include <prefs/input.h>
#include <prefs/overscan.h>
#include <prefs/printergfx.h>
#include <prefs/printertxt.h>
#include <prefs/screenmode.h>
#include <prefs/serial.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>

struct ExecBase *SysBase;
struct Library *DOSBase;
struct Library *IFFParseBase;

static UBYTE *IFFErrTxt[] = {
  "EOF",    /* (end of file, not an error) */
  "EOC",     /* (end of context, not an error) */
  "no lexical scope",
  "insufficient memory",
  "stream read error",
  "stream write error",
  "stream seek error",
  "file corrupt",
  "IFF syntax error",
  "not an IFF file",
  "required call-back hook missing",
  NULL,     /* (return to client, never shown) */
};

LONG main(void)
{
  struct RDArgs *readargs = NULL;
  LONG rargs[2];
  struct IFFHandle *iffhandle;
  struct ContextNode *cnode;
  struct StoredProperty *hdrsp;
  struct StoredProperty *sp;
  UBYTE *filename = NULL;
  LONG ifferror, error = 0, rc = RETURN_OK;

  /* We must set up SysBase (we are not linked with startup code) */
  SysBase = (*((struct Library **) 4));

  /* This no-startup-code example may not be used from Workbench */
```

```
if ((((struct Process *)FindTask(NULL))->pr_CLI)==NULL)
      return(RETURN_FAIL);

if (DOSBase = OpenLibrary("dos.library", 37)) {
  if (IFFParseBase = OpenLibrary ("iffparse.library", 37)) {

    readargs = ReadArgs("FILE/A", rargs, NULL);
    if( (readargs) && (rargs[0]) ) {

      filename = (UBYTE *)rargs[0];

      /* allocate an IFF handle */
      if (iffhandle = AllocIFF()) {
        /* Open the file for reading */
        if (iffhandle->iff_Stream = (LONG)Open(filename, MODE_OLDFILE)) {
          /* initialize the iff handle */
          InitIFFasDOS (iffhandle);
          if ((ifferror = OpenIFF (iffhandle, IFFF_READ)) == 0) {
            PropChunk(iffhandle, ID_PREF, ID_PRHD);

            PropChunk(iffhandle, ID_PREF, ID_FONT);
            PropChunk(iffhandle, ID_PREF, ID_ICTL);
            PropChunk(iffhandle, ID_PREF, ID_INPT);
            PropChunk(iffhandle, ID_PREF, ID_OSCN);
            PropChunk(iffhandle, ID_PREF, ID_PGFX);
            PropChunk(iffhandle, ID_PREF, ID_PTXT);
            PropChunk(iffhandle, ID_PREF, ID_SCRM);
            PropChunk(iffhandle, ID_PREF, ID_SERL);

            for (;;) {
              ifferror = ParseIFF(iffhandle, IFFPARSE_STEP);

               if (ifferror == IFFERR_EOC)
                 continue;
              else if (ifferror)
                break;

              /* Do nothing is this is a PrefHeader chunk,
               * we'll pop it later when there is a pref
               * chunk.
               */
              if (cnode = CurrentChunk(iffhandle))
                if (cnode->cn_ID == ID_PRHD || cnode->cn_ID == ID_FORM)
                  continue;

              /* Get the preferences header, stored previously */
              hdrsp = FindProp(iffhandle, ID_PREF, ID_PRHD);

              if (sp = FindProp(iffhandle, ID_PREF, ID_FONT)) {
                Printf("FrontPen:  %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_FrontPen);
                Printf("BackPen:   %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_BackPen);
                Printf("DrawMode:  %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_DrawMode);
                Printf("Font:      %s\n",
                      (LONG)((struct FontPrefs *)sp->sp_Data)->fp_Name);
                Printf("ta_YSize:  %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_TextAttr.ta_YSize);
                Printf("ta_Style:  %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_TextAttr.ta_Style);
                Printf("ta_Flags:  %ld\n",
                      ((struct FontPrefs *)sp->sp_Data)->fp_TextAttr.ta_Flags);
              } else   if (sp = FindProp(iffhandle, ID_PREF, ID_ICTL)) {
                Printf("TimeOut:   %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_TimeOut);
                Printf("MetaDrag:  %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_MetaDrag);
                Printf("WBtoFront: %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_WBtoFront);
                Printf("FrontToBack: %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_FrontToBack);
                Printf("ReqTrue:    %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_ReqTrue);
                Printf("ReqFalse:    %ld\n",
                      ((struct IControlPrefs *)sp->sp_Data)->ic_ReqFalse);
```

```
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_INPT)) {
                Printf("PointerTicks:       %ld\n",
                      ((struct InputPrefs *)sp->sp_Data)->ip_PointerTicks);
                Printf("DoubleClick/Secs:   %ld\n",
                      ((struct InputPrefs *)sp->sp_Data)->ip_DoubleClick.tv_secs);
                Printf("DoubleClick/Micro: %ld\n",
                      ((struct InputPrefs *)sp->sp_Data)->ip_DoubleClick.tv_micro);
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_OSCN)) {
                Printf("DisplayID:  0x%lx\n",
                      ((struct OverscanPrefs *)sp->sp_Data)->os_DisplayID);
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_PGFX)) {
                Printf("Aspect:     %ld\n",
                      ((struct PrinterGfxPrefs *)sp->sp_Data)->pg_Aspect);
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_PTXT)) {
                Printf("Driver:     %s\n",
                      (LONG)((struct PrinterTxtPrefs *)sp->sp_Data)->pt_Driver);
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_SCRM)) {
                Printf("DisplayID:  0x%lx\n",
                      ((struct ScreenModePrefs *)sp->sp_Data)->sm_DisplayID);
                /* etc */
              } else if (sp = FindProp(iffhandle, ID_PREF, ID_SERL)) {
                Printf("BaudRate:   %ld\n",
                      ((struct SerialPrefs *)sp->sp_Data)->sp_BaudRate);
                /* etc */
              }
            }

            CloseIFF(iffhandle);
          }

          if (ifferror != IFFERR_EOF) {
            rargs[1] = (LONG)IFFErrTxt[-ifferror - 1];
            VFPrintf(Output(), "%s: %s\n", rargs);
            rc = RETURN_FAIL;
          }

          Close(iffhandle->iff_Stream);
        } else
          error = IoErr();

        FreeIFF(iffhandle);
      } else {
        VFPrintf(Output(), "Can't allocate IFF handle\n", NULL);
        rc = RETURN_FAIL;
      }
    } else
      error = IoErr();
    CloseLibrary(IFFParseBase);


    SetIoErr(error);
    if (error) {
      rc = RETURN_FAIL;
      PrintFault(error, filename ? filename : "");
    }
  }

  if(readargs) FreeArgs(readargs);
  CloseLibrary(DOSBase);

} else {
    rc = RETURN_FAIL;
    Write(Output(), "Kickstart 2.0 required\n", 23);
  }

return(rc);
}
```

Function Reference

The following are brief descriptions of the system functions that relate to the use of Preferences. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 13-3: Functions Used with Preferences

Function	Description
GetPrefs()	Old 1.3 (V34) function for making a copy of the **Preferences** structure
SetPrefs()	Old 1.3 (V34) function for overwriting **Preferences** with new data
GetDefPrefs()	Old 1.3 (V34) function for copying default **Preferences** from ROM
StartNotify()	Release 2 DOS library function for monitoring a .prefs file for changes
EndNotify()	Ends notification started with **StartNotify()**
AllocIFF()	IFFParse library function that creates an **IFFHandle** for parsing
InitIFFasDOS()	Initialize the **IFFHandle** as a DOS stream
OpenIFF()	Initialize an **IFFHandle** for reading or writing a new stream
PropChunk()	Specify a property chunk to store
ParseIFF()	Parse an IFF file from the **IFFHandle** stream
CurrentChunk()	Returns the top level context of an IFF stream
FindProp()	Search for a property chunk previously declared with **PropChunk()**
CloseIFF()	Closes an IFF context opened with **OpenIFF()**
FreeIFF()	Frees the **IFFHandle** created with **AllocIFF()**

Chapter 14
WORKBENCH AND ICON LIBRARY

Workbench is the graphic user interface to the Amiga file system that uses symbols called icons to represent disks, directories and files. This chapter shows how to use Workbench and its two support libraries workbench.library and icon.library.

Workbench is both a system program and a screen. Normally it is the first thing the user sees when the machine is booted providing a friendly operating environment for launching applications and performing other important system activities like navigating through the Amiga's hierarchical filing system.

All application programs should be compatible with Workbench. There are only two things you need to know to do this: how to make icons for your application, data files and directories; and how to get arguments if your application is launched from Workbench.

The Info File

The iconic representation of Amiga filing system objects is implemented through *.info* files. In general, for each file, disk or directory that is visible in the Workbench environment, there is an associated *.info* file which contains the icon imagery and other information needed by Workbench.

Icons are associated with a particular file or directory by name. For example, the icon for a file named *myapp* would be stored in a *.info* file named *myapp.info* in the same directory.

To make your application program accessible (and visible) in the Workbench environment, you need only supply a *.info* file with the appropriate name and type. The are four main types of icons (and *.info* files) used to represent Amiga filing system objects (Table 14-1).

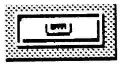
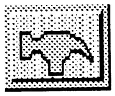
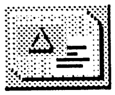
Workbench Icon Type		Filing System Object	Result When Icon Is Activated
Disk		The root level directory	Window opens showing files and subdirectories
Drawer		A subdirectory	Window opens showing files and subdirectories
Tool		An executable file (i.e., an application)	Application runs
Project		A data file	Typically, the application that created the data file runs and the data file is automatically loaded into it.

Table 14-1: Basic Workbench Icon Types

Icons can be created with the IconEdit program (in the Tools directory of the Extras disk), or by copying an existing *.info* file of the correct type. Icons can also be created under program control with **PutDiskObject()**. See the discussion of the icon library functions below for more on this.

For an executable file the icon type must be set to tool. For a data file the icon type must be set to project. Create icons for your application disk and directories too. For a directory, the icon is stored in a *.info* file at the same level where the directory name appears (not in the directory itself). The icon type should be set to drawer. The icon for a disk should always be stored in a file named *disk.info* at the root level directcry of the disk. The icon type should be set to disk. (The icon type can be set and the icon imagery edited with the IconEdit program.)

Workbench Environment

On the Amiga there are at least two ways to start a program running:

- By activating a tool or project icon in Workbench (an icon is activated by pointing to it with the mouse and double-clicking the mouse select button.)
- By typing the name of an executable file at the Shell (also known as the CLI or Command Line Interface)

In the Workbench environment, a program is run as a separate process. A process is simply a task with additional information needed to use DOS library.

By default, a Workbench program does not have a window to which its output will go. Therefore, *stdin* and *stdout* do not point to legal file handles. This means you cannot use *stdio* functions such as **printf()** if your program is started from Workbench unless you first set up a *stdio* window.

Some compilers have options or defaults to provide a *stdio* window for programs started from Workbench. In Release 2, applications can use an auto console window for *stdio* when started from Workbench by opening "CON:0/0/640/200/auto/close/wait" as a file. An auto console window will only open if *stdio* input or output occurs. This can also be handled in the startup code module that comes with your compiler.

ARGUMENT PASSING IN WORKBENCH

Applications started from Workbench receive arguments in the form of a **WBStartup** structure. This is similar to obtaining arguments from a command line interface through **argc** and **argv**. The **WBStartup** message contains an argument count and a pointer to a list of file and directory names.

One Argument

A program started by activating its tool icon gets one argument in the **WBStartup** message: the name of the tool itself.

Two Arguments

All project icons (data files) have a *default tool* field associated with them that tells Workbench which application tool to run in order to operate on the data that the icon represents. When the user activates a project icon, Workbench runs the application specified in the default tool field passing it two arguments in the **WBStartup** message: the name of the tool and the project icon that the user activated.

Multiple Arguments

With extended select, the user can activate many icons at once. (Extended select means the user holds down the Shift key while clicking the mouse select button once on each icon in a group, double-clicking on the last icon.) If one of the icons in a group activated with extended select is an application tool, Workbench runs that application passing it the name of all the other icons in the group. This allows the user to start an application with multiple project files as arguments. If none of the icons in a group activated with extended select is a tool icon, then Workbench looks in the default tool field of each icon in the order they were selected and runs the first tool it finds.

WBSTARTUP MESSAGE

When Workbench loads and starts a program, its sends the program a **WBStartup** message containing the arguments as summarized above. Normally, the startup code supplied with your compiler will place a pointer to **WBStartup** in **argv** for you, set **argc** to zero and call your program.

The **WBStartup** message, whose structure is outlined in *<workbench/startup.h>*, has the following structure elements:

```
struct WBStartup
{
    struct Message      sm_Message;     /* a standard message structure */
    struct MsgPort *    sm_Process;     /* the process descriptor for you */
    BPTR                sm_Segment;     /* a descriptor for your code */
    LONG                sm_NumArgs;     /* the number of elements in ArgList */
    char *              sm_ToolWindow;  /* reserved for future use */
    struct WBArg *      sm_ArgList;     /* the arguments themselves */
};
```

The fields of the **WBStartup** structure are used as follows.

sm_Message
A standard Exec message. The reply port is set to the Workbench.

sm_Process
The process descriptor for the tool (as returned by **CreateProcess()**)

sm_Segment
The loaded code for the tool (returned by **LoadSeg()**)

sm_NumArgs
The number of arguments in **sm_ArgList**

sm_ToolWindow
Reserved (not currently passed in startup message)

sm_ArgList
This is the argument list itself. It is a pointer to an array of **WBArg** structures with **sm_NumArgs** elements.

Workbench arguments are passed as an array of **WBArg** structures in the **sm_ArgList** field of **WBStartup**. The first **WBArg** in the list is always the tool itself. If multiple icons have been selected when a tool is activated, the selected icons are passed to the tool as additional **WBArgs**. If the tool was derived from a default tool, the project will be the second **WBArg**. If extended select was used, arguments other than the tool are passed in the order of selection; the first icon selected will be first (after the tool), and so on.

Each argument is a struct **WBArg** and has two parts: **wa_Name** and **wa_Lock**.

```
struct WBArg
{
    BPTR            wa_Lock;        /* a lock descriptor */
    BYTE *          wa_Name;        /* a string relative to that lock */
};
```

The **wa_Name** element is the name of an AmigaDOS filing system object. The **wa_Name** field of the first **WBArg** is always the name of your program and the **wa_Lock** field is an AmigaDOS **Lock** on the directory where your program is stored.

If your program was started by activating a project icon, then you get a second **WBarg** with the **wa_Name** field containing the file name of the project and the **wa_Lock** containing an AmigaDOS **Lock** on the directory where the project file is stored.

If your program was started through extended select, then you get one **WBArg** for each icon in the selected group in the order they were selected. The **wa_Name** field contains the file name corresponding to each icon unless the icon is for a directory, disk, or the Trashcan in which case the **wa_Name** is set to NULL. The **wa_Lock** field contains an AmigaDOS **Lock** on the directory where the file is stored. (For disk or drawer icons the **wa_Lock** is a lock on the directory represented by the icon. Or, **wa_Lock** may be NULL if the icon type does not support locks.)

Workbench Locks Belong to Workbench. You must never call **UnLock()** on a **wa_Lock**. These locks belong to Workbench, and Workbench will **UnLock()** them when the **WBStartup** message is replied by your startup code. You must also never **UnLock()** your program's initial current directory lock (i.e., the lock returned by an initial **CurrentDir()** call). The classic symptom caused by unlocking Workbench locks is a system hang after your program exits, even though the same program exits with no problems when started from the Shell.

You should save the lock returned from an initial **CurrentDir()**, and **CurrentDir()** back to it before exiting. In the Workbench environment, depending on your startup code, the current directory will generally be set to one of the **wa_Locks**. By using **CurrentDir(wa_Lock)** and then referencing **wa_Name**, you can find, read, and modify the files that have been passed to your program as **WBArgs**.

EXAMPLE OF PARSING WORKBENCH ARGUMENTS

The following example will display all **WBArgs** if started from Workbench, and all Shell arguments if started from the Shell.

```
;/* prargs.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 prargs.c
Blink FROM LIB:c.o,prargs.o LIB LIB:LC.lib,LIB:Amiga.lib TO prargs DEFINE __main=__tinymain
quit
** NOTE: main and tinymain are prepended with two underscores.
**
** PrArgs.c - This program prints all Workbench or Shell (CLI) arguments.
*/
#include <exec/types.h>
#include <workbench/startup.h>
#include <clib/dos_protos.h>
#include <clib/icon_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }   /* Disable SAS Lattice CTRL/C handling */
int chkabort(void) { return(0); }/* really */
#endif

void main(int argc, char **argv)
{
struct WBStartup *argmsg;
struct WBArg *wb_arg;
LONG ktr;
BPTR olddir;
FILE *outFile;

/* argc is zero when run from the Workbench,
**        positive when run from the CLI.
*/
if (argc == 0)
    {
    /* AmigaDOS has a special facility that  allows a window  */
    /* with a console and a file handle to be easily created. */
    /* CON: windows allow you to use fprintf() with no hassle */
    if (NULL != (outFile = fopen("CON:0/0/640/200/PrArgs","r+")))
        {
        /* in SAS/Lattice, argv is a pointer to the WBStartup message
        ** when argc is zero.  (run under the Workbench.)
        */
        argmsg = (struct WBStartup *)argv ;
        wb_arg = argmsg->sm_ArgList ;         /* head of the arg list */

        fprintf(outFile, "Run from the workbench, %ld args.\n",
                         argmsg->sm_NumArgs);
```

```
        for (ktr = 0; ktr < argmsg->sm_NumArgs; ktr++, wb_arg++)
            {
            if (NULL != wb_arg->wa_Lock)
                {
                /* locks supported, change to the proper directory */
                olddir = CurrentDir(wb_arg->wa_Lock) ;

                /* process the file.
                ** If you have done the CurrentDir() above, then you can
                ** access the file by its name.  Otherwise, you have to
                ** examine the lock to get a complete path to the file.
                */
                fprintf(outFile, "\tArg %2.2ld (w/ lock): '%s'.\n",
                                 ktr, wb_arg->wa_Name);

                /* change back to the original directory when done.
                ** be sure to change back before you exit.
                */
                CurrentDir(olddir) ;
                }
            else
                {
                /* something that does not support locks */
                fprintf(outFile, "\tArg %2.2ld (no lock): '%s'.\n",
                                 ktr, wb_arg->wa_Name);
                }
            }
        /* wait before closing down */
        Delay(500L);
        fclose(outFile);
        }
    }
else
    {
    /* using 'tinymain' from lattice c.
    ** define a place to send the output (originating CLI window = "*")
    ** Note - if you open "*" and your program is RUN, the user will not
    ** be able to close the CLI window until you close the "*" file.
    */
    if (NULL != (outFile = fopen("*","r+")))
        {
        fprintf(outFile, "Run from the CLI, %d args.\n", argc);

        for ( ktr = 0; ktr < argc; ktr++)
            {
            /* print an arg, and its number */
            fprintf(outFile, "\tArg %2.2ld: '%s'.\n", ktr, argv[ktr]);
            }
        fclose(outFile);
        }
    }
}
```

The Icon Library

The *.info* file is the center of interaction between applications and Workbench. To help support the Workbench iconic interface and manage *.info* files, the Amiga operating system provides the icon library. The icon library allows you to create icons for data files and directories under program control and examine icons to obtain their Tool Types and other characteristics.

ICON LIBRARY DATA STRUCTURES

The preceding sections discussed how icons are used to pass file name arguments to an application run from the Workbench. Workbench allows other types of arguments to be passed in the Tool Types array of an icon. To examine the Tool Types array or find other characteristics of the icon such as its type, applications need to read in the *.info* file for the icon.

The DiskObject Structure

The actual data present in the *.info* file is organized as a **DiskObject** structure which is defined in the include file *<workbench/workbench.h>*. For a complete listing, see the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. The **DiskObject** structure contains the following elements:

```
struct DiskObject
    {
    UWORD              do_Magic;      /* magic number at start of file */
    UWORD              do_Version;    /* so we can change structure    */
    struct Gadget      do_Gadget;     /* a copy of in core gadget      */
    UBYTE              do_Type;
    char              *do_DefaultTool;
    char             **do_ToolTypes;
    LONG               do_CurrentX;
    LONG               do_CurrentY;
    struct DrawerData *do_DrawerData;
    char              *do_ToolWindow;  /* only applies to tools */
    LONG               do_StackSize;   /* only applies to tools */
    };
```

do_Magic

A magic number that the icon library looks for to make sure that the file it is reading really contains an icon. It should be the manifest constant WB_DISKMAGIC. **PutDiskObject()** will put this value in the structure, and **GetDiskObject** will not believe that a file is really an icon unless this value is correct.

do_Version

This provides a way to enhance the .info file in an upwardly-compatible way. It should be WB_DISKVERSION. The icon library will set this value for you and will not believe weird values.

do_Gadget

This contains all the imagery for the icon. See the "Gadget Structure" section below for more details.

do_Type

The type of the icon; can be set to any of the following values.

WBDISK	The root of a disk
WBDRAWER	A directory on the disk
WBTOOL	An executable program
WBPROJECT	A data file
WBGARBAGE	The Trashcan directory
WBKICK	A Kickstart disk
WBAPPICON	Any object not directly associated with a filing system object, such as a print spooler (new in Release 2).

Table 14-2: Workbench Object Types

do_DefaultTool

Default tools are used for project and disk icons. For projects (data files), the default tool is the program Workbench runs when the project is activated. Any valid AmigaDOS path may be entered in this field such as "SYS:myprogram", "df0:mypaint", "myeditor" or ":work/mytool".

For disk icons, the default tool is the diskcopy program ("SYS:System/DiskCopy") that will be used when *this disk is the source* of a copy.

do_ToolTypes

This is an array of free-format strings. Workbench does not enforce any rules on these strings, but they are useful for passing environment information. See the section on "The ToolTypes Array" below for more information.

do_CurrentX, do_CurrentY

Drawers have a virtual coordinate system. The user can scroll around in this system using the scroll gadgets on the window that opens when the drawer is activated. Each icon in the drawer has a position in the coordinate system. **CurrentX** and **CurrentY** contain the icon's current position in the drawer. Picking a position for a newly created icon can be tricky. NO_ICON_POSITION is a system constant for **do_CurrentX** and **do_CurrentY** that instructs Workbench to pick a reasonable place for the icon. Workbench will place the icon in an unused region of the drawer. If there is no space in the drawers window, the icon will be placed just to the right of the visible region.

do_DrawerData

If the icon is associated with a directory (WBDISK, WBDRAWER, WBGARBAGE), it needs a **DrawerData** structure to go with it. This structure contains an Intuition **NewWindow** structure (see the "Intuition Windows" chapter for more information):

```
struct DrawerData
    {
    struct NewWindow dd_NewWindow;     /* structure to open window      */
    LONG             dd_CurrentX;      /* current x coordinate of origin */
    LONG             dd_CurrentY;      /* current y coordinate of origin */
    };
```

Workbench uses this to hold the current window position and size of the window so it will reopen in the same place.

do_ToolWindow

This field is reserved for future use.

do_StackSize

This is the size of the stack (in bytes) used for running the tool. If this is NULL, then Workbench will use a reasonable default stack size (currently 4K bytes).

Stack Size is Taken from the Project Icon. When a tool is run *via the default tool mechanism* (i.e., a project was activated, not the tool itself), Workbench uses the stack size specified in the project's *.info* file and the tool's *.info* file is ignored.

The Gadget Structure

To hold the icon's image, Workbench uses an Intuition **Gadget** structure, defined in *<intuition/intuition.h>*. Workbench restricts some of the values of the gadget. All unused fields should be set to 0 or NULL. The Intuition gadget structure members that Workbench icons use are listed below.

Gadget Names in Assembly Language Are Different. The assembly language version of the Gadget structure has leading "gg_" for each variable name.

Width

This is the width (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

Height

This is the height (in pixels) of the icon's active region. Any mouse button press within this range will be interpreted as having selected this icon.

Flags

The gadget *must* be of type GADGIMAGE. Three highlight modes are supported: GADGHCOMP, GADGHIMAGE, and GADGBACKFILL. GADGHCOMP complements everything within the area defined by **CurrentX**, **CurrentY**, **Width**, **Height**. GADGHIMAGE uses an alternate selection image. GADGBACKFILL is similar to GADGHCOMP, but ensures that there is no "ring" around the selected image. It does this by first complementing the image, and then flooding all color 3 pixels that are on the border of the image to color 0. All other flag bits should be 0.

Activation

The activation should have only RELVERIFY and GADGIMMEDIATE set.

Type

The gadget type should be BOOLGADGET.

GadgetRender

Set this to an appropriate **Image** structure.

SelectRender

Set this to an appropriate alternate **Image** structure if and only if the highlight mode is GADGHIMAGE.

The **Image** structure is typically the same size as the gadget, except that **Height** is often one pixel less than the gadget height. This allows a blank line between the icon image and the icon name. The image depth *must* be 2; **PlanePick** *must* be 3; and **PlaneOnOff** should be 0. The **NextImage** field should be null.

ICON LIBRARY FUNCTIONS

The icon library functions do all the work needed to read, write and examine an icon's *.info* file and corresponding **DiskObject** structure:

```
struct DiskObject *GetDiskObject(UBYTE *name);
struct DiskObject *GetDiskObjectNew(UBYTE *name);                (V36)
BOOL               PutDiskObject(UBYTE *name, struct DiskObject *diskobj);
void               FreeDiskObject(struct DiskObject *diskobj);
BOOL               DeleteDiskObject(UBYTE *);                    (V37)

UBYTE             *FindToolType(UBYTE **toolTypeArray, UBYTE *typeName);
BOOL               MatchToolValue(UBYTE *typeString, UBYTE *value);

struct DiskObject *GetDefDiskObjectNew(LONG type);               (V36)
BOOL               PutDefDiskObject(struct DiskObject *diskobj); (V36)

UBYTE             *BumpRevision(UBYTE *newbuf, UBYTE *oldname);
```

The icon library routine **GetDiskObject()** reads an icon's *.info* file from disk into a **DiskObject** structure it creates in memory where it can be examined or altered. **PutDiskObject()** writes the **DiskObject** out to disk and **FreeDiskObject()** frees the memory it used. If you modify any pointers in a DiskObject acquired via **GetDiskObject()**, replace the old pointers before calling **FreeDiskObject()** so that the proper memory will be freed.

Release 2 includes a new function named **GetDiskObjectNew()** that works the same as **GetDiskObject()**, except that if no *.info* file is found, a default **DiskObject** will be created for you. Also new for Release 2 is **DeleteDiskObject()** for removing *.info* files from disk, and the functions **GetDefDiskObject()** and **PutDefDiskObject()** which allow the default icons in ROM to be copied or replaced with new defaults in RAM.

Once an icon's *.info* file has been read into a **DiskObject** structure, the functions **FindToolType()** and **MatchToolType()** can be used to examine the icon's Tool Types array.

THE TOOL TYPES ARRAY

Earlier sections discussed how Workbench passes filenames as arguments to a program that's about to run. Workbench also allows other types of arguments to be passed in the Tool Types array of an icon. The Tool Types array is found in the **do_ToolTypes** field of the icon's **DiskObject** structure.

In brief, Tool Types is an array of pointers to strings that contain any information an application wants to store such as the program options that were in effect when the icon was created. These strings can be used to encode information which will be available to all applications that read the icon's *.info* file. Users can enter and change a selected icon's Tool Types by choosing Information in the Workbench Icons menu.

Workbench does not place many restrictions on the Tool Types array, but there are a few conventions you should follow. A string may be no more than 128 bytes long. The alphabet used is 8-bit ANSI (for example, normal ASCII with foreign-language extensions). This means that users may enter Tool Type strings containing international characters. Avoid special or nonprinting characters. The case of the characters is currently significant, so the string "Window" is not equal to "WINDOW".

The general format for a Tool Types entry is <name>=<value>[|<value>], where <name> is the field name and <value> is the text to associate with that name. Multiple values for one name may be separated by a vertical bar. The values may be the type of the file, programs that can access the data, parameters to be passed to an application, etc. For example, a paint program might set:

    FILETYPE = PaintProgram | ILBM

This Tool Type indicates that the file is an ILBM, perhaps with some additional chunks of data specific to PaintProgram.

Tool Type strings have few restrictions but there are some reserved Tool Types that are parsed by Workbench itself when an application is started from an icon. The reserved Tool Types are **TOOLPRI=n** (sets the Exec task priority at which Workbench will start the application), **STARTPRI=n** (sets the starting order for icons in the Wbstartup drawer), and **DONOTWAIT** (tells Workbench not to wait for the return of a program started via an icon in the Wbstartup drawer). In addition to the reserved Tool Types, which applications should not use, there are standard Tool Types, which applications should use only in the standard way. For a list of standard Tool Types refer to the *Amiga User Interface Style Guide*.

Two routines are provided to help you deal with the Tool Types array. **FindToolType()** returns the value of a Tool Type element. Using the above example, if you are looking for FILETYPE, the string "PaintProgram|ILBM" will be returned. **MatchToolValue()** returns nonzero if the specified string is in the reference value string. This routine knows how to parse vertical bars. For example, using the reference value strings of "PaintProgram" or "ILBM", **MatchToolValue()** will return TRUE for "ILBM" and "PaintProgram" and FALSE for everything else.

EXAMPLE OF READING ICONS AND PARSING TOOL TYPES

The following example demonstrates icon creation, icon reading and Tool Type parsing in the Workbench environment. When called from the Shell, the example creates a small data file in RAM: and creates or updates a project icon for the data file. The created project icon points to this example as its default tool. When the new project icon is double-clicked, Workbench will invoke the default tool (this example) as a Workbench process, and pass it a description of the project data file as a Workbench argument (**WBArg**) in the **WBStartup** message.

```
;/* iconexample.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 iconexample.c
Blink FROM LIB:c.o,iconexample.o TO iconexample LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** iconexample.c - Workbench icon startup, creation, and parsing example
*/

#include <exec/types.h>
#include <libraries/dos.h>
#include <workbench/workbench.h>
#include <workbench/startup.h>

#include <clib/alib_protos.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/icon_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }   /* Disable SAS Lattice CTRL/C handling */
int chkabort(void) { return(0); }/* really */
#endif

/* our functions */
void cleanexit(UBYTE *,LONG);
void cleanup(void);
void message(UBYTE *);
BOOL makeIcon(UBYTE *, char **, char *);
BOOL showToolTypes(struct WBArg *);

UBYTE *projname     = "RAM:Example_Project";
UBYTE *conwinname   = "CON:10/10/620/180/iconexample";

UBYTE deftoolname[] = {"iconexample"};

USHORT IconImageData1[] =  {
/* Plane 0 */
    0x0000,0x0000,0x0000,0x1000,0x0000,0x0000,0x0000,0x3000,
    0x0FFF,0xFFFC,0x0000,0x3000,0x0800,0x0004,0x0000,0x3000,
    0x0800,0x07FF,0xFFC0,0x3000,0x08A8,0xA400,0x00A0,0x3000,
    0x0800,0x0400,0x0090,0x3000,0x08AA,0xA400,0x0088,0x3000,
    0x0800,0x042A,0xA0FC,0x3000,0x082A,0xA400,0x0002,0x3000,
    0x0800,0x0400,0x0002,0x3000,0x0800,0xA42A,0xA0A2,0x3000,
    0x0800,0x0400,0x0002,0x3000,0x0950,0xA42A,0x8AA2,0x3000,
    0x0800,0x0400,0x0002,0x3000,0x082A,0xA400,0x0002,0x3000,
    0x0800,0x042A,0x2AA2,0x3000,0x0FFF,0xFC00,0x0002,0x3000,
    0x0000,0x0400,0x0002,0x3000,0x0000,0x07FF,0xFFFE,0x3000,
    0x0000,0x0000,0x0000,0x3000,0x7FFF,0xFFFF,0xFFFF,0xF000,
/* Plane 1 */
    0xFFFF,0xFFFF,0xFFFF,0xE000,0xD555,0x5555,0x5555,0x4000,
    0xD000,0x0001,0x5555,0x4000,0xD7FF,0xFFF9,0x5555,0x4000,
    0xD7FF,0xF800,0x0015,0x4000,0xD757,0x5BFF,0xFF55,0x4000,
    0xD7FF,0xFBFF,0xFF65,0x4000,0xD755,0x5BFF,0xFF75,0x4000,
    0xD7FF,0xFBD5,0x5F01,0x4000,0xD7D5,0x5BFF,0xFFFD,0x4000,
    0xD7FF,0xFBFF,0xFFFD,0x4000,0xD7FF,0x5BD5,0x5F5D,0x4000,
    0xD7FF,0xFBFF,0xFFFD,0x4000,0xD6AF,0x5BD5,0x755D,0x4000,
    0xD7FF,0xFBFF,0xFFFD,0x4000,0xD7D5,0x5BFF,0xFFFD,0x4000,
```

```
    0xD7FF,0xFBD5,0xD55D,0x4000,0xD000,0x03FF,0xFFFD,0x4000,
    0xD555,0x53FF,0xFFFD,0x4000,0xD555,0x5000,0x0001,0x4000,
    0xD555,0x5555,0x5555,0x4000,0x8000,0x0000,0x0000,0x0000,
};

struct Image iconImage1 =
    {
    0, 0,                   /* Top Corner */
    52, 22, 2,              /* Width, Height, Depth */
    &IconImageData1[0],     /* Image Data */
    0x003, 0x000,           /* PlanePick,PlaneOnOff */
    NULL                    /* Next Image */
    };

UBYTE *toolTypes[] =
    {
    "FILETYPE=text",
    "FLAGS=BOLD|ITALICS",
    NULL
    };

struct DiskObject projIcon =
    {
    WB_DISKMAGIC,                       /* Magic Number */
    WB_DISKVERSION,                     /* Version */
        {                               /* Embedded Gadget Structure */
        NULL,                           /* Next Gadget Pointer */
        97,12,52,23,                    /* Left,Top,Width,Height */
        GADGIMAGE|GADGHBOX,             /* Flags */
        GADGIMMEDIATE|RELVERIFY,        /* Activation Flags */
        BOOLGADGET,                     /* Gadget Type */
        (APTR)&iconImage1,              /* Render Image */
        NULL,                           /* Select Image */
        NULL,                           /* Gadget Text */
        NULL,                           /* Mutual Exclude */
        NULL,                           /* Special Info */
        0,                              /* Gadget ID */
        NULL                            /* User Data */
        },
    WBPROJECT,                          /* Icon Type */
    deftoolname,                        /* Default Tool */
    toolTypes,                          /* Tool Type Array */
    NO_ICON_POSITION,                   /* Current X */
    NO_ICON_POSITION,                   /* Current Y */
    NULL,                               /* Drawer Structure */
    NULL,                               /* Tool Window */
    4000                                /* Stack Size */
    };

/* Opens and allocations we must clean up */
struct Library *IconBase = NULL;
FILE *conwin = NULL;
LONG olddir = -1;

BOOL FromWb;

void main(int argc, char **argv)
    {
    struct WBStartup *WBenchMsg;
    struct WBArg *wbarg;
    FILE  *file;
    LONG  wLen;
    SHORT i;

    FromWb = (argc==0) ? TRUE : FALSE;

    /* Open icon.library */
    if(!(IconBase = OpenLibrary("icon.library",33)))
         cleanexit("Can't open icon.library\n",RETURN_FAIL);

    /* If started from CLI, this example will create a small text
     * file RAM:Example_Project, and create an icon for the file
     * which points to this program as its default tool.
     */
    if(!FromWb)
        {
```

```
        /* Make a sample project (data) file */
        wLen = -1;
        if(file=fopen(projname,"w"))
            {
            wLen = fprintf(file,"Have a nice day\n");
            fclose(file);
            }
        if(wLen < 0) cleanexit("Error writing data file\n",RETURN_FAIL);

        /* Now save/update icon for this data file */
        if(makeIcon(projname, toolTypes, deftoolname))
             {
             printf("%s data file and icon saved.\n",projname);
             printf("Use Workbench menu Icon Information to examine the icon.\n");
             printf("Then copy this example (iconexample) to RAM:\n");
             printf("and double-click the %s project icon\n",projname);
             }
        else cleanexit("Error writing icon\n",RETURN_FAIL);
        }

    else  /* Else we are FromWb - ie. we were either
           * started by a tool icon, or as in this case,
           * by being the default tool of a project icon.
           */
        {
        if(!(conwin = fopen(conwinname,"r+")))
             cleanexit("Can't open output window\n",RETURN_FAIL);

        WBenchMsg = (struct WBStartup *)argv;

        /* Note wbarg++ at end of FOR statement steps through wbargs.
         * First arg is our executable (tool).  Any additional args
         * are projects/icons passed to us via either extend select
         * or default tool method.
         */
        for(i=0, wbarg=WBenchMsg->sm_ArgList;
            i < WBenchMsg->sm_NumArgs;
            i++, wbarg++)
            {
            /* if there's a directory lock for this wbarg, CD there */
            olddir = -1;
            if((wbarg->wa_Lock)&&(*wbarg->wa_Name))
                olddir = CurrentDir(wbarg->wa_Lock);

            showToolTypes(wbarg);

            if((i>0)&&(*wbarg->wa_Name))
                fprintf(conwin,"In Main. We could open the %s file here\n",
                                 wbarg->wa_Name);
            if(olddir != -1)  CurrentDir(olddir); /* CD back where we were */
            }
        Delay(500);
        }
    cleanup();
    exit(RETURN_OK);
    }

BOOL makeIcon(UBYTE *name, char **newtooltypes, char *newdeftool)
    {
    struct DiskObject *dobj;
    char *olddeftool;
    char **oldtooltypes;
    BOOL success = FALSE;

    if(dobj=GetDiskObject(name))
        {
        /* If file already has an icon, we will save off any fields we
         * need to update, update those fields, put the object, restore
         * the old field pointers and then free the object.  This will
         * preserve any custom imagery the user has, and the user's
         * current placement of the icon.  If your application does
         * not know where the user currently keeps your application,
         * you should not update his dobj->do_DefaultTool.
         */
         oldtooltypes = dobj->do_ToolTypes;
         olddeftool = dobj->do_DefaultTool;
```

```
        dobj->do_ToolTypes = newtooltypes;
        dobj->do_DefaultTool = newdeftool;

        success = PutDiskObject(name,dobj);

        /* we must restore the original pointers before freeing */
        dobj->do_ToolTypes = oldtooltypes;
        dobj->do_DefaultTool = olddeftool;
        FreeDiskObject(dobj);
        }
    /* Else, put our default icon */
    if(!success)  success = PutDiskObject(name,&projIcon);
    return(success);
    }

BOOL showToolTypes(struct WBArg *wbarg)
    {
    struct DiskObject *dobj;
    char **toolarray;
    char *s;
    BOOL success = FALSE;

    fprintf(conwin,"\nWBArg Lock=0x%lx, Name=%s\n",
                           wbarg->wa_Lock,wbarg->wa_Name);

    if((*wbarg->wa_Name) && (dobj=GetDiskObject(wbarg->wa_Name)))
        {
        fprintf(conwin,"  We have read the DiskObject (icon) for this arg\n");
        toolarray = (char **)dobj->do_ToolTypes;

        if(s=(char *)FindToolType(toolarray,"FILETYPE"))
            {
            fprintf(conwin,"    Found tooltype FILETYPE with value %s\n",s);
            }
        if(s=(char *)FindToolType(toolarray,"FLAGS"))
            {
            fprintf(conwin,"    Found tooltype FLAGS with value %s\n",s);
            if(MatchToolValue(s,"BOLD"))
                fprintf(conwin,"      BOLD flag requested\n");
            if(MatchToolValue(s,"ITALICS"))
                fprintf(conwin,"      ITALICS flag requested\n");
            }
        /* Free the diskobject we got */
        FreeDiskObject(dobj);
        success = TRUE;
        }
    else if(!(*wbarg->wa_Name))
        fprintf(conwin,"  Must be a disk or drawer icon\n");
    else
        fprintf(conwin,"  Can't find any DiskObject (icon) for this WBArg\n");
    return(success);
    }


/* Workbench-started programs with no output window may want to display
 * messages in a different manner (requester, window title, etc)
 */
void message(UBYTE *s)
    {
    if(FromWb && conwin)  fprintf(conwin,s,strlen(s));
    else if (!FromWb) printf(s);
    }

void cleanexit(UBYTE *s, LONG n)
    {
    if(*s)  message(s);
    cleanup();
    exit(n);
    }

void cleanup()
   {
   if(conwin)     fclose(conwin);
   if(IconBase)   CloseLibrary(IconBase);
   }
```

The Workbench Library

Workbench arguments are sent to an application when it is started. There are also special facilities in Release 2 of Workbench that allow an application that is already running to get additional arguments. These special facilities are known as **AppWindow**, **AppIcon** and **AppMenuItem**.

An **AppWindow** is a special kind of window that allows the user to drag icons into it. Applications that set up an **AppWindow** will receive a message from Workbench whenever the user moves an icon into the **AppWindow**. The message contains the name of the file or directory that the icon represents.

An **AppIcon** is similar to an **AppWindow**. It is a special type of icon that allows the user to drag other icons on top of it. Like **AppWindows**, an application that sets up an **AppIcon** will receive a message from Workbench whenever the user moves another icon on top of the **AppIcon**. The message contains the name of the file or directory that the moved icon represents.

An **AppMenuItem** allows an application to add a custom menu item to the usual set of menu choices supported by Workbench. An application that sets up an **AppMenuItem** will receive a message from Workbench whenever the user picks that item from the Workbench menus.

When an application receives the messages described above, the message will include **struct WBArg *am_ArgList** containing the names (**wa_Name**) and directory locks (**wa_Lock**) of all selected icons that were passed as arguments by the user. This **am_ArgList** has the same format as the **sm_ArgList** of a **WBStartup** message.

WORKBENCH LIBRARY FUNCTIONS

AppWindows, **AppIcons** and **AppMenuItems** extend the user's ability to perform operations with the Workbench iconic interface. They all provide graphical methods for passing arguments to a running application. In order to manage **AppWindows**, **AppIcons** and **AppMenuItems**, the Amiga OS includes these Workbench library functions:

```
struct AppIcon     *AddAppIconA( ULONG, ULONG, char *, struct MsgPort *, struct FileLock *,
                                 struct DiskObject *, struct *TagItem );
struct AppMenuItem *AddAppMenuItemA( ULONG, ULONG, char *, struct MsgPort *,
                                     struct *TagItem);
struct AppWindow   *AddAppWindowA( ULONG, ULONG, struct Window *, struct MsgPort *,
                                   struct *TagItem);

BOOL                RemoveAppIcon(struct AppIcon *);
BOOL                RemoveAppMenuItem(struct AppMenuItem *);
BOOL                RemoveAppWindow(struct AppWindow  *);
```

The functions **AddAppMenuItemA()**, **AddAppWindowA()** and **AddAppIconA()** have alternate entry points using the same function name without the trailing **A**. The alternate functions accept any **TagItem** arguments on the stack instead of from an array. See the listings below for examples.

AN APPICON EXAMPLE

The example listed here shows how to create an **AppIcon** and obtain arguments from Workbench when the user drops other icons on top of it. The **AppIcon** will appear as a disk icon named "TestAppIcon" on the Workbench screen. (All **AppIcons** appear on the Workbench screen or window.)

For convenience, this example code uses **GetDefDiskObject()** to create the icon imagery for the **AppIcon**. Applications should never do this. Use your own custom imagery for **AppIcons** instead.

```
/* appicon.c - Compiled under SAS C 5.10 with lc -L appicon.c            */
/* Requires Kickstart version 37 or later.  Works from the Shell (CLI) only */

#include <exec/types.h>          /* Need this for the Amiga variable types  */
#include <workbench/workbench.h> /* This has DiskObject and AppIcon structs */
#include <workbench/startup.h>   /* This has WBStartup and WBArg structs    */
#include <exec/libraries.h>      /* Need this to check library versions     */

#include <clib/icon_protos.h>    /* Icon (DiskObject) function prototypes   */
#include <clib/exec_protos.h>    /* Exec message, port and library functions*/
#include <clib/wb_protos.h>      /* AppIcon function protos                 */

#ifdef LATTICE
int CXBRK(void) { return(0); }   /* Disable SAS Lattice CTRL/C handling */
int chkabort(void) { return(0); }/* really */
#endif

extern struct Library *SysBase;
struct Library *IconBase;
struct Library *WorkbenchBase;

void main(int argc, char **argv)
{
struct DiskObject   *dobj=NULL;
struct MsgPort     *myport=NULL;
struct AppIcon    *appicon=NULL;
struct AppMessage *appmsg=NULL;

LONG dropcount=0L;
ULONG x;
BOOL success=0L;

/* Get the the right version of the Icon Library, initialize IconBase */
if(IconBase = OpenLibrary("icon.library",37))
  {
  /* Get the the right version of the Workbench Library */
  if (WorkbenchBase=OpenLibrary("workbench.library",37))
    {
    /* This is the easy way to get some icon imagery */
    /* Real applications should use custom imagery   */
    dobj=GetDefDiskObject(WBDISK);
    if(dobj!=0)
      {
      /* The type must be set to NULL for a WBAPPICON */
      dobj->do_Type=NULL;

      /* The CreateMsgPort() function is in Exec version 37 and later only */
      myport=CreateMsgPort();
      if(myport)
        {
        /* Put the AppIcon up on the Workbench window */
        appicon=AddAppIconA(0L,0L,"TestAppIcon",myport,NULL,dobj,NULL);
        if(appicon)
          {
          /* For the sake of this example, we allow the AppIcon */
          /* to be activated only five times.                   */
          printf("Drop files on the Workbench AppIcon\n");
          printf("Example exits after 5 drops\n");

          while(dropcount<5)
            {
            /* Here's the main event loop where we wait for */
            /* messages to show up from the AppIcon         */
```

```
          WaitPort(myport);

          /* Might be more than one message at the port... */
          while(appmsg=(struct AppMessage *)GetMsg(myport))
            {
            if(appmsg->am_NumArgs==0L)
              {
              /* If NumArgs is 0 the AppIcon was activated directly */
              printf("User activated the AppIcon.\n");
              printf("A Help window for the user would be good here\n");
              }
            else if(appmsg->am_NumArgs>0L)
              {
              /* If NumArgs is >0 the AppIcon was activated by */
              /* having one or more icons dropped on top of it */
              printf("User dropped %ld icons on the AppIcon\n",
                                              appmsg->am_NumArgs);
              for(x=0;x<appmsg->am_NumArgs;x++)
                {
                printf("#%ld name='%s'\n",x+1,appmsg->am_ArgList[x].wa_Name);
                }
              }
            /* Let Workbench know we're done with the message */
            ReplyMsg((struct Message *)appmsg);
            }
          dropcount++;
          }
        success=RemoveAppIcon(appicon);
        }
      /* Clear away any messages that arrived at the last moment */
      while(appmsg=(struct AppMessage *)GetMsg(myport))
          ReplyMsg((struct Message *)appmsg);
      DeleteMsgPort(myport);
      }
    FreeDiskObject(dobj);
    }
  CloseLibrary(WorkbenchBase);
  }
 CloseLibrary(IconBase);
 }
}
```

AN APPMENUITEM EXAMPLE

This example shows how to create an **AppMenuItem**. The example adds a menu item named "Browse Files" to the Workbench Tools menu. (All **AppMenuItems** appear in the Workbench Tools menu.) When the menu item is activated, the example program receives a message from Workbench and then attempts to start up an instance of the More program. (The More program is in the Utilities directory of the Workbench disk.)

The example starts up the More program as a separate, asynchronous process using the new **SystemTags()** function of Release 2 AmigaDOS. For more about the **SystemTags()** function refer to the *AmigaDOS Manual, 3rd Edition* from Bantam Books. When the **AppMenuItem** has been activated five times, the program exits after freeing any system resources it has used.

```
/* appmenuitem.c - Compiled under SAS C 5.10 with lc -L appmenuitem.c      */
/* Requires Kickstart version 37 or later.  Works from the Shell (CLI) only */

#include <exec/types.h>         /* Need this for the Amiga variable types  */
#include <workbench/workbench.h> /* This has DiskObject and AppIcon structs */
#include <workbench/startup.h>   /* This has WBStartup and WBArg structs    */
#include <exec/libraries.h>
#include <dos/dostags.h>
#include <stdio.h>
#include <clib/dos_protos.h>
#include <clib/exec_protos.h>    /* Exec message, port and library functions*/
#include <clib/wb_protos.h>      /* AppMenuItem function protos             */
```

```
#ifdef LATTICE
int CXBRK(void) { return(0); }   /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }/* really */
#endif

extern struct Library *SysBase;
struct Library *WorkbenchBase;

void main(int argc, char **argv)
{
struct MsgPort      *myport=NULL;
struct AppMenuItem *appitem=NULL;
struct AppMessage   *appmsg=NULL;
LONG result, x, count=0L;
BOOL success=0L;
BPTR file;

if (WorkbenchBase = OpenLibrary("workbench.library",37))
  {
  /* The CreateMsgPort() function is in Exec version 37 and later only */
  if(myport = CreateMsgPort())
    {
    /* Add our own AppMenuItem to the Workbench Tools Menu */
    appitem=AddAppMenuItemA(0L,                     /* Our ID# for item */
                    (ULONG)"SYS:Utilities/More",  /* Our UserData     */
                           "Browse Files",        /* MenuItem Text    */
                            myport,NULL);         /* MsgPort, no tags */
    if(appitem)
      {
      printf("Select Workbench Tools demo menuitem 'Browse Files'\n");

      /* For this example, we allow the AppMenuItem to be selected */
      /* only once, then we remove it and exit                     */
      WaitPort(myport);
      while((appmsg=(struct AppMessage *)GetMsg(myport)) && (count<1))
        {
        /* Handle messages from the AppMenuItem - we have only one  */
        /* item so we don't have to check its appmsg->am_ID number. */
        /* We'll System() the command string that we passed as     */
        /* userdata when we added the menu item.                    */
        /* We find our userdata pointer in appmsg->am_UserData      */

        printf("User picked AppMenuItem with %ld icons selected\n",
                                                appmsg->am_NumArgs);
        for(x=0;x<appmsg->am_NumArgs;x++)
           printf("  #%ld name='%s'\n",x+1,appmsg->am_ArgList[x].wa_Name);

        count++;
        if( file=Open("CON:0/40/640/150/AppMenu Example/auto/close/wait",
                        MODE_OLDFILE)  )      /* for any stdio output */
          {
          result=SystemTags((UBYTE *)appmsg->am_UserData,SYS_Input,file,
                                                        SYS_Output,NULL,
                                                        SYS_Asynch,TRUE,
                                                        TAG_DONE);
          /* If Asynch System() itself fails, we must close file */
          if(result == -1) Close(file);
          }
        ReplyMsg((struct Message *)appmsg);
        }
      success=RemoveAppMenuItem(appitem);
      }

    /* Clear away any messages that arrived at the last moment */
    /* and let Workbench know we're done with the messages     */
    while(appmsg=(struct AppMessage *)GetMsg(myport))
      {
      ReplyMsg((struct Message *)appmsg);
      }
    DeleteMsgPort(myport);
    }
  CloseLibrary(WorkbenchBase);
  }
}
```

AN APPWINDOW EXAMPLE

This example shows how to create an **AppWindow** and obtain arguments from Workbench when the user drops an icon into it. The **AppWindow** will appear on the Workbench screen with the name "AppWindow" and will run until the window's close gadget is selected. If any icons are dropped into the **AppWindow**, the program prints their arguments in the Shell window.

```
/* appwindow.c - Compiled under SAS C 5.10 with lc -L appwindow.c         */
/* Requires Kickstart version 37 or later.  Works from the Shell (CLI) only */

#include <exec/types.h>         /* Need this for the Amiga variable types  */
#include <workbench/workbench.h> /* This has DiskObject and AppWindow      */
#include <workbench/startup.h>   /* This has WBStartup and WBArg structs   */
#include <exec/libraries.h>      /* Need this to check library versions    */

#include <stdio.h>

#include <clib/intuition_protos.h>
#include <clib/exec_protos.h>
#include <clib/wb_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }    /* Disable SAS Lattice CTRL/C handling */
int chkabort(void) { return(0); }/* really */
#endif

struct Library      *IntuitionBase;
struct Library      *WorkbenchBase;

void main(int argc, char **argv)
{
  struct MsgPort *awport;
  struct Window  *win;
  struct AppWindow *appwin;
  struct IntuiMessage *imsg;
  struct AppMessage *amsg;
  struct WBArg   *argptr;

  ULONG           winsig, appwinsig, signals, id = 1, userdata = 0;
  BOOL            done = FALSE;
  int             i;

  if (IntuitionBase = OpenLibrary("intuition.library", 37))
    {
    if (WorkbenchBase = OpenLibrary("workbench.library", 37))
      {
      /* The CreateMsgPort() function is in Exec version 37 and later only */
      if (awport = CreateMsgPort())
         {
         if (win = OpenWindowTags(NULL,
                                 WA_Width, 200,        WA_Height, 50,
                                 WA_IDCMP, CLOSEWINDOW,
                                 WA_Flags, WINDOWCLOSE | WINDOWDRAG,
                                 WA_Title, "AppWindow",
                                 TAG_DONE))
          {
          if (appwin = AddAppWindow(id, userdata, win, awport, NULL))
            {
            printf("AppWindow added... Drag files into AppWindow\n");
            winsig    = 1L << win->UserPort->mp_SigBit;
            appwinsig = 1L << awport->mp_SigBit;

            while (! done)
              {
              /* Wait for IDCMP messages and AppMessages */
              signals = Wait( winsig | appwinsig );

              if(signals & winsig)      /* Got an IDCMP message */
                {
                while (imsg = (struct IntuiMessage *) GetMsg(win->UserPort))
                  {
                  if (imsg->Class = CLOSEWINDOW)    done = TRUE;
                  ReplyMsg((struct Message *) imsg);
```

```
            }
          }
        if(signals & appwinsig)    /* Got an AppMessage */
          {
          while (amsg = (struct AppMessage *) GetMsg(awport))
            {
            printf("AppMsg: Type=%ld, ID=%ld, NumArgs=%ld\n",
                     amsg->am_Type, amsg->am_ID, amsg->am_NumArgs);
            argptr = amsg->am_ArgList;
            for (i = 0; i < amsg->am_NumArgs; i++)
              {
              printf("   arg(%ld): Name='%s', Lock=%lx\n",
                      i, argptr->wa_Name, argptr->wa_Lock);
              argptr++;
              }
            ReplyMsg((struct Message *) amsg);
            }
          }
        }     /* done */
      RemoveAppWindow(appwin);
      }
    CloseWindow(win);
    }
  /* Make sure there are no more outstanding messages */
  while(amsg = (struct AppMessage *)GetMsg(awport))
        ReplyMsg((struct Message *)amsg);
  DeleteMsgPort(awport);
  }
 CloseLibrary(WorkbenchBase);
 }
CloseLibrary(IntuitionBase);
}
}
```

Workbench and the Startup Code Module

Standard startup code handles the detail work of interfacing with the arguments and environment of Workbench and the Shell (or CLI). This section describes the behavior of standard startup modules such as the ones supplied with SAS (Lattice) C and Manx Aztec C.

The environment for a program started from Workbench is quite different from the environment for a program started from the Shell. The Shell does not create a new process for a program; it jumps to the program's code and the program shares the process with the Shell. Programs run under the Shell have access to all the Shell's environment, including the ability to modify that environment. (Programs run from the Shell should be careful to restore all values that existed on startup.) Workbench starts a program as a new DOS process, explicitly passing the execution environment to the program.

WORKBENCH STARTUP

When the user activates a project or tool icon, the program is run as a separate process asynchronous to Workbench. This allows the user to take full advantage of the multitasking features of the Amiga. A process is simply a task with additional information needed to use DOS library.

When Workbench loads and starts a program, its sends the program a **WBStartup** message containing the arguments as described earlier. The **WBStartup** also contains a pointer to the new **Process** structure which describes the execution environment of the program. The **WBStartup** message is posted to the message port of the program's **Process** structure.

The **Process** message port is for the exclusive use of DOS, so this message must be removed from the port before using any DOS library functions. Normally this is handled by the startup code module that comes

with your compiler so you don't have to worry about this unless you are writing your own startup code.

Standard startup code modules also set up **SysBase**, the pointer to the Exec master library, and open the DOS library setting up **DOSBase**. That is why Exec and AmigaDOS functions can be called by C applications without first opening a library; the startup code that applications are linked with handles this. Some special startups may also set up NIL: input and output streams, or may open a *stdio* window so that the Workbench applications can use stdio functions such as **printf()P.**

The startup code can tell if it is running in the Workbench environment because the **pr_CLI** field of the **Process** structure will contain NULL. In that case the startup code removes the **WBStartup** message from the **Process** message port with **GetMsg()** before using any functions in the DOS library.

> *Do Not Use the Process Message Port for Anything Else.* The message port in a **Process** structure is for the exclusive use of the DOS library.

Standard startup code will pass the **WBStartup** message pointer in **argv** and 0 (zero) in **argc** if the program is started from Workbench. These values are pushed onto the stack, and the startup code calls the application code that it is linked with as a function. When the application code exits back to the startup code, the startup code closes and frees all opens and allocations it made. It will then **Forbid()**, and **ReplyMsg()** the **WBStartup** message, notifying Workbench that the application **Process** may be terminated and its code unloaded from memory.

> *Avoid the DOS Exit() function.* The DOS **Exit()** function does *not* return an application to the startup code that called it. If you wish to exit your application, use the exit function provided by your startup code (usually lower-case **exit()**, or **_exit** for assembler), passing it a valid DOS return code as listed in the include file *<libraries/dos.h>*.

SHELL STARTUP

When a program is started from the Shell (or a Shell script), standard startup modules will parse the command line (received in A0, with length in D0) into an array of pointers to individual argument strings placing them in **argv**, and an argument count in **argc**.

If a program is started from the Shell, **argc** will always equal at least one and the first element in **argv** will always be a pointer to the command name. Other command line arguments are stored in turn. For example, if the command line was:

```
df0:myprogram  "my file1"  file2  ;this is a comment
```

then **argc** will be 3, **argv**[0] will be "df0:myprogram", **argv**[1] will be "my file1", and **argv**[2] will be "file2". Correct startup code will strip spaces between arguments and trailing spaces from the last argument and will also properly deal with quoted arguments with embedded spaces.

As with Workbench, standard startup code for the Shell sets up **SysBase**, the pointer to the Exec master library, and opens the DOS library setting up **DOSBase**. C applications that are linked with standard startup code can call an Exec or AmigaDOS functions without opening the library first.

The startup code also fills in the *stdio* file handles (_stdin, _stdout, etc.) for the application. Finally **argv** and **argc**, are pushed onto the stack and the application is called via a JSR. When the application returns or exits back to the startup code, the startup code closes and frees all opens and allocations it has made for the application, and then returns to the system with the whatever value the program exited with.

Link your applications only with standard, tested startup code of some type such as the module supplied with your compiler. Startup code provides your programs with correct, consistent handling of Shell command line and Workbench arguments and will perform some initializations and cleanups which would otherwise need to be handled by your own code. Very small startups can be used for programs that do not require command line arguments.

A few words of warning for those of you who do not use standard startup code:

- If you are started as a Workbench process, you *must* **GetMsg()** the **WBStartup** message before using any functions in the DOS library.
- You *must* turn off task switching (with **Forbid()**) before replying the **WBStartup** message from Workbench. This will prevent Workbench from unloading your code before you can exit properly.
- If you do your own command line parsing, you *must* provide the user with consistent and correct handling of command line arguments.

Function Reference

The following are brief descriptions of the functions in workbench.library and icon.library. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 14-3: Icon Library Functions

Function	Description
GetDiskObject()	Read the .info file of an icon into a **DiskObject** structure
GetDiskObjectNew()	Same as **GetDiskObject()** but returns a default icon if none exists
PutDiskObject()	Write a **DiskObject** structure to disk as a .info file
FreeDiskObject()	Free the **DiskObject** structure created by **GetDiskObject()**
DeleteDiskObject()	Deletes a given .info file from disk
FindToolType()	Return the value of an entry in the icon's Tool Type array
MatchToolValue()	Check a Tool Type entry against a given value
GetDefDiskObject()	Read the default icon for a given icon type
PutDefDiskObject()	Replace the default icon for a given icon type (V36)
AddFreeList()	Add memory you have allocated to a FreeList
FreeFreeList()	Free all the memory for entries in the FreeList
BumpRevision()	Create a new name for a second copy of a Workbench object

Table 14-4: Workbench Library Functions

Function	Description
AddAppIcon()	Add an **AppIcon** to Workbench
AddAppMenuItem()	Add an **AppMenuItem** to the Workbench Tools menu
AddAppWindow()	Add an **AppWindow** to Workbench
RemoveAppIcon()	Remove an **AppIcon** to Workbench
RemoveAppMenuItem()	Remove an **AppMenuItem** to the Workbench Tools menu
RemoveAppWindow()	Remove an **AppWindow** to Workbench

Chapter 15
GADTOOLS LIBRARY

GadTools is a new library in Release 2 that is designed to simplify the task of creating user interfaces with Intuition. GadTools offers a flexible and varied selection of gadgets and menus to help programmers through what used to be a difficult chore.

Intuition, the Amiga's graphical user interface, is a powerful and flexible environment. It allows a software designer a great degree of flexibility in creating dynamic and powerful user interfaces. However, the drawback of this flexibility is that programming even straightforward user interfaces can be complicated, and certainly difficult for first-time Intuition programmers.

What the Gadget Toolkit (GadTools) attempts to do is harness the power of Intuition by providing easy-to-use, high-level chunks of user interface. GadTools doesn't pretend to answer all possible user interface needs of every application but by meeting the user interface needs of most applications, GadTools greatly simplifies the problem of designing user-friendly software on the Amiga. (For applications with special needs, custom solutions can be created with Intuition's already-familiar gadgets or its new Boopsi object-oriented custom gadget system; GadTools is compatible with these.)

Elements of GadTools

GadTools is the easy way to program gadgets and menus. With GadTools, the system handles the detail work required to control gadgets and menus so the application uses less code and simpler data structures.

Another key benefit of GadTools is its standardized and elegant look. All applications that use GadTools will share a similar appearance and behavior. Users will appreciate a sense of instant familiarity even the first time they use a product.

GadTools provides a significant degree of visual consistency across multiple applications that use it. For instance, in Release 2, the Preferences editors, the Workbench ''Information'' window and Commodities Exchange share the same polished look and feel thanks to GadTools. There is also internal consistency between different elements of GadTools; the look is clean and orderly. Depth is used not just for visual embellishment, but as an important cue. For instance, the user is free to select symbols that appear inside a ''raised'' area, but ''recessed'' areas are informational only, and clicking in them has no effect.

GadTools is not amenable to creative post-processing or hacking by programmers looking to achieve a result other than what GadTools currently offers. Software developers whose needs extend beyond the standard features of GadTools should create custom gadgets that share the look and feel of GadTools by using either BOOPSI or by directly programming gadgets at a lower level. See the chapters on "Intuition Gadgets" and "BOOPSI" for more information. Follow the GadTools rules. Only in this way may GadTools grow and improve without hindrance, even allowing new features to automatically appear in future software when reasonable.

GADTOOLS TAGS

Many of the GadTools functions use **TagItem** arrays or *tag lists* to pass information across the function interface. These tag-based functions come in two types, one that takes a pointer to an array of tag items and one that takes a variable number of tag item arguments directly in the function call. In general, the second form, often called the *varargs* form because the call takes a variable number of arguments, is provided for convenience and is internally converted to the first form. When looking through the Autodocs or other Amiga reference material, the documentation for both forms is usually available in the array-based function description.

All GadTools tags begin with a leading "GT". In general, they also have a two-letter mnemonic for the kind of gadget in question. For example, slider gadgets recognize tags such as "GTSL_Level". The GadTools tags are defined in *<libraries/gadtools.h>*. Certain GadTools gadgets also recognize other Intuition tags such as GA_Disabled and PGA_Freedom, which can be found in *<intuition/gadgetclass.h>*.

For more information on tags and tag-based functions, be sure to see the "Utility Library" chapter in this manual.

GadTools Menus

GadTools menus are easy to use. Armed only with access to a **VisualInfo** data structure, GadTools allows the application to easily create, layout and delete Intuition menus.

Normally, the greatest difficulty in creating menus is that a large number of structures must be filled out and linked. This is bothersome since much of the required information is orderly and is easier to do algorithmically than to do manually. GadTools handles this for you.

There are also many complexities in creating a sensible layout for menus. This includes some mechanical items such as handling various font sizes, automatic columnization of menus that are too tall and accounting for space for checkmarks and Amiga-key equivalents. There are also aesthetic considerations, such as how much spacing to provide, where sub-menus should be placed and so on.

GadTools menu functions support all the features that most applications will need. These include:

- An easily constructed and legible description of the menus.
- Font-sensitive layout.
- Support for menus and sub-menus.

- Sub-menu indicators (a ">>" symbol attached to items with sub-menus).
- Separator bars for sectioning menus.
- Command-key equivalents.
- Checkmarked and mutually exclusive checkmarked menu items.
- Graphical menu items.

With GadTools, it takes only one structure, the **NewMenu** structure, to specifiy the whole menu bar, For instance, here is how a typical menu strip containing two menus might be specified:

```
struct NewMenu mynewmenu[] =
    {
        { NM_TITLE, "Project",    0 , 0, 0, 0,},
        {  NM_ITEM, "Open...",   "O", 0, 0, 0,},
        {  NM_ITEM, "Save",      "S", 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Print",      0 , 0, 0, 0,},
        {   NM_SUB, "Draft",      0 , 0, 0, 0,},
        {   NM_SUB, "NLQ",        0 , 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Quit...",   "Q", 0, 0, 0,},

        { NM_TITLE, "Edit",       0 , 0, 0, 0,},
        {  NM_ITEM, "Cut",       "X", 0, 0, 0,},
        {  NM_ITEM, "Copy",      "C", 0, 0, 0,},
        {  NM_ITEM, "Paste",     "V", 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Undo",      "Z", 0, 0, 0,},

        {   NM_END, NULL,         0 , 0, 0, 0,},
    };
```

This **NewMenu** specification would produce the two menus below:

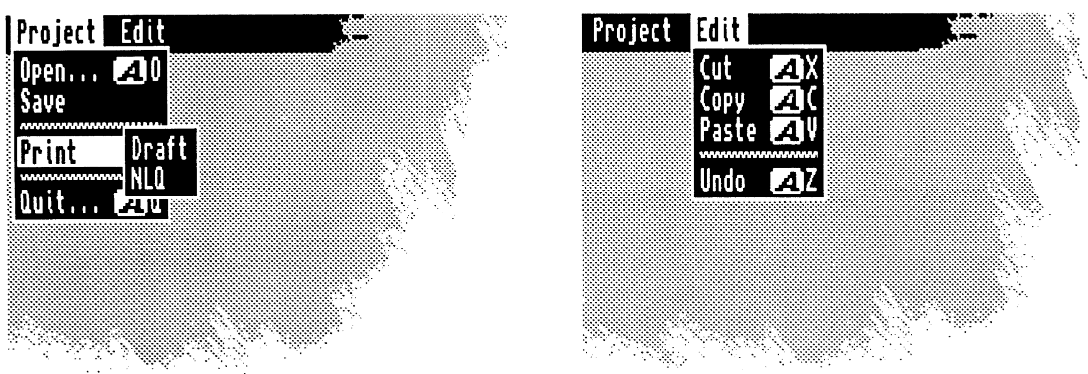


Figure 15-1: Two Example Menus

The **NewMenu** arrays are designed to be read easily. The elements in the **NewMenu** array appear in the same order as they will appear on-screen. Unlike the lower-level menu structures described in the "Intuition Menus" chapter earlier, there is no need to specify sub-menus first, then the menu items with their sub-menus, and finally the menu headers with their menu items. The indentation shown above also helps highlight the relationship between menus, menu items and sub-items.

THE NEWMENU STRUCTURE

The **NewMenu** structure used to specify GadTools menus is defined in *<libraries/gadtools.h>* as follows:

```
struct NewMenu
    {
    UBYTE nm_Type;
    STRPTR nm_Label;
    STRPTR nm_CommKey;
    UWORD nm_Flags;
    LONG nm_MutualExclude;
    APTR nm_UserData;
    };
```

nm_Type

The first field, **nm_Type**, defines what this particular **NewMenu** describes. The defined types provide an unambiguous and convenient representation of the application's menus.

NM_TITLE

Used to signify a textual menu heading. Each NM_TITLE signifies the start of a new menu within the menu strip.

NM_ITEM or **IM_ITEM**

Used to signify a textual (NM_ITEM) or graphical (IM_ITEM) menu item. Each NM_ITEM or IM_ITEM becomes a menu item in the current menu.

NM_SUB or **IM_SUB**

Used to signify a textual (NM_SUB) or graphical (IM_SUB) menu sub-item. All the consecutive NM_SUBs and IM_SUBs that follow a menu item (NM_ITEM or IM_ITEM) compose that item's sub-menu. A subsequent NM_ITEM or IM_ITEM would indicate the start of the next item in the original menu, while a subsequent NM_TITLE would begin the next menu.

NM_END

Used to signify the end of the **NewMenu** structure array. The last element of the array must have NM_END as its type.

nm_Label

NM_TITLE, NM_ITEM and NM_SUB are used for textual menu headers, menu items and sub-items respectively, in which case **nm_Label** points to the string to be used. This string is not copied, but rather a pointer to it is kept. Therefore the string must remain valid for the active life of the menu.

Menus don't have to use text, GadTools also supports graphical menu items and sub-items (graphical menu headers are not possible since they are not supported by Intuition). Simply use IM_ITEM and IM_SUB instead and point **nm_Label** at a valid **Image** structure. The **Image** structure can contain just about any graphic image (see the chapter on "Intuition Images, Line Drawing and Text" for more on this).

Sometimes it is a good idea to put a separator between sets of menu items or sub-items. The application may want to separate drastic menu items such as "Quit" or "Delete" from more mundane ones. Another good idea is to group related checkmarked items by using separator bars.

NM_BARLABEL

GadTools will provide a separator bar if the special constant NM_BARLABEL is supplied for the **nm_Label** field of an NM_ITEM or NM_SUB.

nm_CommKey

A single character string used as the Amiga-key equivalent for the menu item or sub-item.

Menu headers cannot have command keys. Note that assigning a command-key equivalent to a menu item that has sub-items is meaningless and should be avoided.

The **nm_CommKey** field is a pointer to a string and not a character itself. This was done in part because routines to support different languages typically return strings, not characters. The first character of the string is actually copied into the resulting **MenuItem** structure.

nm_Flags

The **nm_Flags** field of the **NewMenu** structure corresponds roughly to the **Flags** field of the Intuition's lower-level **Menu** and **MenuItem** structures.

For programmer convenience the sense of the Intuition MENUENABLED and ITEMENABLED flags are inverted. When using GadTools, menus, menu items and sub-items are enabled by default.

NM_MENUDISABLED

To specify a disabled menu, set the NM_MENUDISABLED flag in this field.

NM_ITEMDISABLED

To disable an item or sub-item, set the NM_ITEMDISABLED flag.

The Intuition flag bits COMMSEQ (indication of a command-key equivalent), ITEMTEXT (indication of a textual or graphical item) and HIGHFLAGS (method of highlighting) will be automatically set depending on other attributes of the menus. Do not set these values in **nm_Flags**.

The **nm_Flags** field is also used to specify checkmarked menu items. To get a checkmark that the user can toggle, set the CHECKIT and MENUTOGGLE flags in the **nm_Flags** field. Also set the CHECKED flag if the item or sub-item is to start in the checked state.

nm_MutualExclude

For specifying mutual exclusion of checkmarked items. All the items or sub-items that are part of a mutually exclusive set should have the CHECKIT flag set.

This field is a bit-wise representation of the items (or sub-items), in the same menu or sub-menu, that are excluded by this item (or sub-item). In the simple case of mutual exclusion, where each choice excludes all others, set **nm_MutualExclude** to **~(1<<item number)** or **~1**, **~2**, **~4**, **~8**, etc. Separator bars count as items and should be included in the position calculation. See the "Intuition Menus" chapter for more details on menu mutual exclusion.

nm_UserData

The **NewMenu** structure also has a user data field. This data is stored with the Intuition **Menu** or **MenuItem** structures that GadTools creates. Use the macros **GTMENU_USERDATA(menu)** and **GTMENUITEM_USERDATA(menuitem)** defined in *<libraries/gadtools.h>* to extract or change the user data fields of menus and menu items, respectively.

The application may place index numbers in this field and perform a **switch** statement on them, instead of using the Intuition menu numbers. The advantage of this is that the numbers chosen remain valid even if the menus are rearranged, while the Intuition menu numbers would change when the menus are rearranged.

Alternately, an efficient technique for menu handling is to create a handler function for each menu item and put a pointer to that function in the corresponding item's **UserData** field. When the program receives a IDCMP_MENUPICK message it may call the selected item's function through this field.

GADTOOLS MENUS EXAMPLE

The functions used to set up and control GadTools menus are discussed in the next section. Before looking at these functions in detail, it may be helpful to look at a brief example.

```
/* gadtoolsmenu.c
** Example showing the basic usage of the menu system with a window.
** Menu layout is done with GadTools, as is recommended for applications.
**
** Compiled with SAS C v5.10a
** lc -b1 -cfistq -v -y gadtoolsmenu
** blink FROM LIB:c.o gadtoolsmenu.o TO gadtoolsmenu LIB LIB:lc.lib LIB:amiga.lib
*/

#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <libraries/gadtools.h>

#include <clib/exec_protos.h>
#include <clib/gadtools_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

struct Library *GadToolsBase;
struct IntuitionBase *IntuitionBase;

struct NewMenu mynewmenu[] =
    {
        { NM_TITLE, "Project",    0 , 0, 0, 0,},
        {  NM_ITEM, "Open...",   "O", 0, 0, 0,},
        {  NM_ITEM, "Save",      "S", 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Print",      0 , 0, 0, 0,},
        {   NM_SUB, "Draft",      0 , 0, 0, 0,},
        {   NM_SUB, "NLQ",        0 , 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Quit...",   "Q", 0, 0, 0,},

        { NM_TITLE, "Edit",       0 , 0, 0, 0,},
        {  NM_ITEM, "Cut",       "X", 0, 0, 0,},
        {  NM_ITEM, "Copy",      "C", 0, 0, 0,},
```

```
        {  NM_ITEM, "Paste",      "V", 0, 0, 0,},
        {  NM_ITEM, NM_BARLABEL,  0 , 0, 0, 0,},
        {  NM_ITEM, "Undo",       "Z", 0, 0, 0,},

        {   NM_END, NULL,          0 , 0, 0, 0,},
    };


/*
** Watch the menus and wait for the user to select the close gadget
** or quit from the menus.
*/
VOID handle_window_events(struct Window *win, struct Menu *menuStrip)
{
struct IntuiMessage *msg;
SHORT done;
UWORD menuNumber;
UWORD menuNum;
UWORD itemNum;
UWORD subNum;
struct MenuItem *item;

done = FALSE;
while (FALSE == done)
    {
    /* we only have one signal bit, so we do not have to check which
    ** bit broke the Wait().
    */
    Wait(1L << win->UserPort->mp_SigBit);

    while ( (FALSE == done) &&
            (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort))))
        {
        switch (msg->Class)
            {
            case IDCMP_CLOSEWINDOW:
                done = TRUE;
                break;
            case IDCMP_MENUPICK:
                menuNumber = msg->Code;
                while ((menuNumber != MENUNULL) && (!done))
                    {
                    item = ItemAddress(menuStrip, menuNumber);

                    /* process the item here! */
                    menuNum = MENUNUM(menuNumber);
                    itemNum = ITEMNUM(menuNumber);
                    subNum  = SUBNUM(menuNumber);

                    /* stop if quit is selected. */
                    if ((menuNum == 0) && (itemNum == 5))
                        done = TRUE;

                    menuNumber = item->NextSelect;
                    }
                break;
            }
        ReplyMsg((struct Message *)msg);
        }
    }
}


/*
** Open all of the required libraries and set-up the menus.
*/
VOID main(int argc, char *argv[])
{
struct Window *win;
APTR *my_VisualInfo;
struct Menu *menuStrip;

/* Open the Intuition Library */
IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", 37);
if (IntuitionBase != NULL)
    {
```

```
    /* Open the gadtools Library */
    GadToolsBase = OpenLibrary("gadtools.library", 37);
    if (GadToolsBase != NULL)
        {
        if (NULL != (win = OpenWindowTags(NULL,
                            WA_Width,  400,        WA_Activate,    TRUE,
                            WA_Height, 100,        WA_CloseGadget, TRUE,
                            WA_Title,  "Menu Test Window",
                            WA_IDCMP,  IDCMP_CLOSEWINDOW | IDCMP_MENUPICK,
                            TAG_END)))
            {
            if (NULL != (my_VisualInfo = GetVisualInfo(win->WScreen, TAG_END)))
                {
                if (NULL != (menuStrip = CreateMenus(mynewmenu, TAG_END)))
                    {
                    if (LayoutMenus(menuStrip, my_VisualInfo, TAG_END))
                        {
                        if (SetMenuStrip(win, menuStrip))
                            {
                            handle_window_events(win,menuStrip);

                            ClearMenuStrip(win);
                            }
                        FreeMenus(menuStrip);
                        }
                    }
                FreeVisualInfo(my_VisualInfo);
                }
            CloseWindow(win);
            }
        CloseLibrary((struct Library *)GadToolsBase);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}
```

FUNCTIONS FOR GADTOOLS MENUS

In this section the basic GadTools menu functions are presented. See the listing above for an example of how to use these functions.

Creating Menus

The **CreateMenus()** function takes an array of **NewMenus** and creates a set of initialized and linked Intuition **Menu, MenuItem, Image** and **IntuiText** structures, that need only to be formatted before being used. Like the other tag-based functions, there is a **CreateMenusA()** call that takes a pointer to an array of **TagItems** and a **CreateMenus()** version that expects to find its tags on the stack.

```
struct Menu *CreateMenusA( struct NewMenu *newmenu, struct TagItem *taglist );
struct Menu *CreateMenus( struct NewMenu *newmenu, Tag tag1, ... );
```

The first argument to these functions, **newmenu**, is a pointer to an array of **NewMenu** structures as described earlier. The tag arguments can be any of the following items:

GTMN_FrontPen (ULONG)
The pen number to use for menu text and separator bars. The default is zero.

GTMN_FullMenu (BOOL)
(New for V37, ignored under V36). This tag instructs **CreateMenus()** to fail if the supplied **NewMenu** structure does not describe a complete **Menu** structure. This is useful if the application does not have direct control over the **NewMenu** description, for example if it has user-configurable menus. The default is FALSE.

GTMN_SecondaryError (ULONG *)

(New for V37, ignored under V36). This tag allows **CreateMenus()** to return some secondary error codes. Supply a pointer to a NULL-initialized ULONG, which will receive an appropriate error code as follows:

GTMENU_INVALID

Invalid menu specification. For instance, a sub-item directly following a menu-title or an incomplete menu. **CreateMenus()** failed in this case, returning NULL.

GTMENU_NOMEM

Failed for lack of memory. **CreateMenus()** returned NULL.

GTMENU_TRIMMED

The number of menus, items or sub-items exceeded the maximum number allowed so the menu was trimmed. In this case, **CreateMenus()** does not fail but returns a pointer to the trimmed **Menu** structure.

NULL

If no error was detected.

CreateMenus() returns a pointer to the first **Menu** structure created, while all the **MenuItem** structures and any other **Menu** structures are attached through the appropriate pointers. If the **NewMenu** structure begins with an entry of type NM_ITEM or IM_ITEM, then **CreateMenus()** will return a pointer to the first **MenuItem** created, since there will be no first **Menu** structure. If the creation fails, usually due to a lack of memory, **CreateMenus()** will return NULL.

Starting with V37, GadTools will not create any menus, menu items or sub-items in excess of the maximum number allowed by Intuition. Up to 31 menus may be defined, each menu with up to 63 items, each item with up to 31 sub-items. See the "Intuition Menus" chapter for more information on menus and their limitations. If the **NewMenu** array describes a menu that is too big, **CreateMenus()** will return a trimmed version. GTMN_SecondaryError can be used to learn when this happens.

Menus need to be added to the window with Intuition's **SetMenuStrip()** function. Before doing this, they must be formatted with a call to **LayoutMenus()**.

Layout of the Menus

The **Menu** and **MenuItem** structures returned by **CreateMenus()** contain no size or positional information. This information is added in a separate layout step, using **LayoutMenus()**. As with the other tag-based functions, the program may call either **LayoutMenus()** or **LayoutMenusA()**.

```
BOOL LayoutMenusA( struct Menu *firstmenu, APTR vi, struct TagItem *taglist );
BOOL LayoutMenus( struct Menu *firstmenu, APTR vi, Tag tag1, ... );
```

Set **firstmenu** to a pointer to a **Menu** structure returned by a previous call to **CreateMenus()**. The **vi** argument is a a **VisualInfo** handle obtained from **GetVisualInfo()**. See the documentation of GadTools gadgets below for more about this call. For the tag arguments, **tag1** or **taglist**, **LayoutMenus()** recognizes a single tag:

GTMN_TextAttr

A pointer to an openable font (**TextAttr** structure) to be used for the menu item and sub-item text. The default is to use the screen's font.

LayoutMenus() fills in all the size, font and position information for the menu strip. **LayoutMenus()** returns TRUE if successful and FALSE if it fails. The usual reason for failure is that the font supplied cannot be opened.

LayoutMenus() takes care of calculating the width, height and position of each individual menu item and sub-item, as well as the positioning of all menus and sub-menus. In the event that a menu would be too tall for the screen, it is broken up into multiple columns. Additionally, whole menus may be shifted left from their normal position to ensure that they fit on screen. If a large menu is combined with a large font, it is possible, even with columnization and shifting, to create a menu too big for the screen. GadTools does not currently trim off excess menus, items or sub-items, but relies on Intuition to clip menus at the edges of the screen.

It is perfectly acceptable to change the menu layout by calling **ClearMenuStrip()** to remove the menus, then **LayoutMenus()** to make the change and then **SetMenuStrip()** to display the new layout. Do this when changing the menu's font (this can be handled by a tag to **LayoutMenus()**), or when updating the menu's text (to a different language, for instance). Run-time language switching in menus will be discussed later.

Layout for Individual Menus

LayoutMenuItems() performs the same function as **LayoutMenus()**, but only affects the menu items and sub-items of a single menu instead of the whole menu strip. Ordinarily, there is no need to call this function after having called **LayoutMenus()**. This function is useful for adding menu items to an extensible menu, such as the Workbench "Tools" menu.

For example, a single **MenuItem** can be created by calling **CreateMenus()** with a two-entry **NewMenu** array whose first entry is of type NM_ITEM and whose second is of type NM_END. The menu strip may then be removed and this new item linked to the end of an extensible menu by placing its address in the **NextItem** field of the last **MenuItem** in the menu. **LayoutMenuItems()** can then be used to to recalculate the layout of just the items in the extensible menu and, finally, the menu strip can be reattached to the window.

```
BOOL LayoutMenuItemsA( struct MenuItem *firstitem, APTR vi, struct TagItem *taglist );
BOOL LayoutMenuItems( struct MenuItem *firstitem, APTR vi, Tag tag1, ... );
```

Set **firstitem** to a pointer to the first **MenuItem** in the linked list of **MenuItems** that make up the **Menu**. (See the "Intuition Menus" chapter for more about these structures.) Set **vi** to the address of a **VisualInfo** handle obtained from **GetVisualInfo()**. The tag arguments, **tag1** or **taglist**, may be set as follows:

GTMN_TextAttr
A pointer to an openable font (**TextAttr** structure) to be used for the menu item and sub-item text. The default is to use the screen's font.

GTMN_Menu
Use this tag to provide a pointer to the **Menu** structure whose **FirstItem** is passed as the first parameter to this function. This tag should always be used.

LayoutMenuItems() returns TRUE if it succeeds and FALSE otherwise.

Freeing Menus

The **FreeMenus()** function frees all the memory allocated by the corresponding call to **CreateMenus()**.

```
void FreeMenus( struct Menu *menu );
```

Its one argument is the **Menu** or **MenuItem** pointer that was returned by **CreateMenus()**. It is safe to call **FreeMenus()** with a NULL parameter, the function will then return immediately.

GADTOOLS MENUS AND INTUIMESSAGES

If the window uses GadTools menus and GadTools gadgets, then use the **GT_GetIMsg()** and **GT_ReplyIMsg()** functions described below (or **GT_FilterIMsg()** and **GT_PostFilterIMsg()**, if applicable). However, if the window has GadTools menus, but no GadTools gadgets, it is acceptable to use **GetMsg()** and **ReplyMsg()** in the usual manner.

Additionally, no context need be created with **CreateContext()** if no GadTools gadgets are used. For more about these functions, see the section on "Other GadTools Functions" later in this chapter.

RESTRICTIONS ON GADTOOLS MENUS

GadTools menus are regular Intuition menus. Once the menus have been laid out, the program may do anything with them, including attaching them or removing them from windows, enabling or disabling items, checking or unchecking checkmarked menu items, etc. See the documentation for **SetMenuStrip()**, **ClearMenuStrip()**, **ResetMenuStrip()**, **OnMenu()** and **OffMenu()** in the "Intuition Menus" chapter for full details.

If a GadTools-created menu strip is not currently attached to any window, the program may change the text in the menu headers (**Menu->MenuName**), the command-key equivalents (**MenuItem->Command**) or the text or imagery of menu items and sub-items, which can be reached as:

```
((struct IntuiText *)MenuItem->ItemFill)->IText
```

or

```
((struct Image *)MenuItem->ItemFill)
```

The application may also link in or unlink menus, menu items or sub-items. However, do not add sub-items to a menu item that was not created with sub-items and do not remove all the sub-items from an item that was created with some.

Any of these changes may be made, provided the program subsequently calls **LayoutMenus()** or **LayoutMenuItems()** as appropriate. Then, reattach the menu strip using **SetMenuStrip()**.

Some of these manipulations require walking the menu strip using the usual Intuition-specified linkages. Beginning with the first **Menu** structure, simply follow its **FirstItem** pointer to get to the first **MenuItem**. The **MenuItem->SubItem** pointer will lead to the sub-menus. **MenuItems** are connected via the **MenuItem->NextItem** field. Successive menus are linked together with the **Menu->NextMenu** pointer. Again, see the "Intuition Menus" chapter for details.

LANGUAGE-SENSITIVE MENUS

Allowing the application to switch the language displayed in the menus, can be done quite easily. Simply detach the menu strip and replace the strings in the **IntuiText** structures as described above. It may be convenient to store some kind of index number in the **Menu** and **MenuItem UserData** which can be used to retrieve the appropriate string for the desired language. After all the strings have been installed, call **LayoutMenus()** and **SetMenuStrip()**.

If the application has the localized strings when the menus are being created, it simply places the pointers to the strings and command shortcuts into the appropriate fields of the **NewMenu** structure. The menus may then be processed in the normal way.

GadTools Gadgets

The heart of GadTools is in its ability to easily create and manipulate a sophisticated and varied array of gadgets. GadTools supports the following kinds of gadgets:

Table 15-1: Standard Gadget Types Supported by the GadTools Library

Gadget Type	Description or Example Usage
Button	Familiar action gadgets, such as "OK" or "Cancel".
String	For text entry.
Integer	For numeric entry.
Checkboxes	For on/off items.
Mutually exclusive	Radio buttons, select one choice among several.
Cycle	Multiple-choice, pick one of a small number of choices.
Sliders	To indicate a level within a range.
Scrollers	To indicate a position in a list or area.
Listviews	Scrolling lists of text.
Palette	Color selection.
Text-display	Read-only text.
Numeric-display	Read-only numbers.

GadTools gadget handling consists of a body of routines to create, manage and delete any of the 12 kinds of standard gadgets listed in table 15-1, such as buttons, sliders, mutually exclusive buttons and scrolling lists.

To illustrate the flexibility, power and simplicity that GadTools offers, consider the GadTools slider gadget. This gadget is used to indicate and control the level of something, for example volume, speed or color intensity. Without GadTools, applications have to deal directly with Intuition proportional and their arcane variables, such as **HorizBody** to control the slider knob's size and **HorizPot** to control the knob's position. Using the GadTools slider allows direct specification of the minimum and maximum levels of the slider, as well as its current level. For example, a color slider might have a minimum level of 0, a maximum level of 15 and a current level of 11.

To simplify event-processing for the slider, GadTools only sends the application a message when the knob has moved far enough to cause the slider level, as expressed in application terms, to change. If a user were to slowly drag the knob of this color slider all the way to the right, the program will only hear messages for levels 12, 13, 14 and 15, with an optional additional message when the user releases the mouse-button.

Changing the current level of the slider from within the program is as simple as specifying the new level in a function call. For instance, the application might set the slider's value to 5.

As a final point, the slider is very well-behaved. When the user releases the mouse-button, the slider immediately snaps to the centered position for the level. If a user sets their background color to light gray, which might have red = green = blue = 10, all three color sliders will have their knobs at precisely the same relative position, instead of anywhere in the range that means "ten".

THE NEWGADGET STRUCTURE

For most gadgets, the **NewGadget** structure is used to specify its common attributes. Additional attributes that are unique to specific kinds of gadgets are specified as tags sent to the **CreateGadget()** function (described below).

The **NewGadget** structure is defined in *<intuition/gadtools.h>* as:

```
struct NewGadget
    {
    WORD ng_LeftEdge, ng_TopEdge;
    WORD ng_Width, ng_Height;
    UBYTE *ng_GadgetText;
    struct TextAttr *ng_TextAttr;
    UWORD ng_GadgetID;
    ULONG ng_Flags;
    APTR ng_VisualInfo;
    APTR ng_UserData;
    };
```

The fields of the **NewGadget** structure are used as follows:

ng_LeftEdge, ng_TopEdge
Define the position of the gadget being created.

ng_Width and **ng_Height**
Define the size of the gadget being created.

ng_GadgetText
Most gadgets have an associated label, which might be the text in a button or beside a checkmark. This field contains a pointer to the appropriate string. Note that only the pointer to the text is copied, the text itself is not. The string supplied must remain constant and valid for the life of the gadget.

ng_TextAttr
The application must specify a font to use for the label and any other text that may be associated with the gadget.

ng_Flags
Used to describe general aspects of the gadget, which includes where the label is to be placed and whether the label should be rendered in the highlight color. The label may be positioned on the left side, the right side, centered above, centered below or dead-center on the gadget. For most gadget kinds, the label is placed on the left side by default, exceptions will be noted.

ng_GadgetID, ng_UserData
These user fields are copied into the resulting **Gadget** structure.

ng_VisualInfo

This field must contain a pointer to an instance of the **VisualInfo** structure, which contains information needed to create and render GadTools gadgets. The **VisualInfo** structure itself is private to GadTools and subject to change. Use the specialized GadTools functions for accessing the **VisualInfo** pointer, defined below. Never access or modify fields within this structure.

CREATING GADGETS

The main call used to create a gadget with GadTools is **CreateGadget()**. This function can be used to create a single gadget or it can be called repeatedly to create a linked list of gadgets. It takes three arguments followed by a set of tags:

```
struct Gadget *CreateGadget( ULONG kind, struct Gadget *prevgad, struct NewGadget *newgad,
                             struct TagItem *taglist)
struct Gadget *CreateGadgetA(ULONG kind, struct Gadget *prevgad, struct NewGadget *newgad,
                             struct Tag tag1, ...)
```

Set the **kind** argument to one of the 12 gadget types supported by GadTools. Set the **prevgad** argument to the gadget address returned by **CreateContext()** if this is the first (or only) gadget in the list. Subsequent calls to **CreateGadget()** can be used to create and link gadgets together in a list in which case the **prevgad** argument is set to the address of the gadget returned by the preceding call to **CreateGadget()**.

Set the **newgad** argument to the address of the **NewGadget** structure describing the gadget to be created and set any special attributes for this gadget type using the tag arguments, **tag1** or **taglist**. For instance, the following code fragment might be used to create the color slider discussed earlier:

```
slidergad = CreateGadget(SLIDER_KIND, newgadget, prevgad,
    GTSL_Min, 0,
    GTSL_Max, 15,
    GTSL_Level, 11,
    TAG_END);
```

CreateGadget() typically allocates and initializes all the necessary Intuition structures, including in this case the **Gadget**, **IntuiText** and **PropInfo** structures, as well as certain buffers. For more about these underlying structures, see the "Intuition Gadgets" chapter.

Since **CreateGadget()** is a tag-based function, it is easy to add more tags to get a fancier gadget. For example, GadTools can optionally display the running level beside the slider. The caller must supply a **printf()**-style formatting string and the maximum length that the string will resolve to when the number is inserted:

```
slidergad = CreateGadget(SLIDER_KIND, newgadget, prevgad,
    GTSL_Min, 0,
    GTSL_Max, 15,
    GTSL_Level, 11,
    GTSL_LevelFormat, "%2ld" /* printf()-style formatting string */
    GTSL_MaxLevelLen, 2,     /* maximum length of string         */
    TAG_END);
```

The level, 0 to 15 in this example, would then be displayed beside the slider. The formatting string could instead be "%2ld/15", so the level would be displayed as "0/15" through "15/15".

HANDLING GADGET MESSAGES

GadTools gadgets follow the same input model as other Intuition components. When the user operates a GadTools gadget, Intuition notifies the application about the input event by sending an **IntuiMessage**. The application can get these messages at the **Window.UserPort**. However GadTools gadgets use different message handling functions to get and reply these messages. Instead of the Exec functions **GetMsg()** and **ReplyMsg()**, applications should get and reply these messages through a pair of special GadTools functions, **GT_GetIMsg()** and **GT_ReplyIMsg()**.

```
struct IntuiMessage *GT_GetIMsg(struct MsgPort *iport)
void  GT_ReplyIMsg(struct IntuiMessage *imsg)
```

For **GT_GetIMsg()**, the **iport** argument should be set to the window's **UserPort**. For **GT_ReplyIMsg()**, the **imsg** argument should be set to a pointer to the **IntuiMessage** returned by **GT_GetIMsg()**.

These functions ensure that the application only sees the gadget events that concern it and in a desirable form. For example, with a GadTools slider gadget, a message only gets through to the application when the slider's level actually changes and that level can be found in the **IntuiMessage**'s **Code** field:

```
imsg = GT_GetIMsg(win->UserPort);
object = imsg->IAddress;
class = imsg->Class;
code = imsg->Code;
GT_ReplyIMsg(imsg);
switch (class)
    {
    case IDCMP_MOUSEMOVE:
        if (object == slidergad)
            {
            printf("Slider at level %ld0, code);
            }
        ...
        break;
    ...
    }
```

In general, the **IntuiMessages** received from GadTools contain more information in the **Code** field than is found in regular Intuition gadget messages. Also, when dealing with GadTools a lot of messages (mostly IDCMP_MOUSEMOVEs) do not have to be processed by the application. These are two reasons why dealing with GadTools gadgets is much easier than dealing with regular Intuition gadgets. Unfortunately this processing cannot happen magically, so applications must use **GT_GetIMsg()** and **GT_ReplyIMsg()** where they would normally have used **GetMsg()** and **ReplyMsg()**.

GT_GetIMsg() actually calls **GetMsg()** to remove a message from the specified window's **UserPort**. If the message pertains to a GadTools gadget then some dispatching code in GadTools will be called to process the message. What the program will receive from **GT_GetIMsg()** is actually a copy of the real **IntuiMessage**, possibly with some supplementary information from GadTools, such as the information typically found in the **Code** field.

The **GT_ReplyIMsg()** call will take care of cleaning up and replying to the real **IntuiMessage**.

> *Warning:* When an IDCMP_MOUSEMOVE message is received from a GadTools gadget, GadTools arranges to have the gadget's pointer in the **IAddress** field of the **IntuiMessage**. While this is extremely convenient, it is also untrue of messages from regular Intuition gadgets (described in the ''Intuition Gadgets'' chapter). Do not make the mistake of assuming it to be true.

This description of the inner workings of **GT_GetIMsg()** and **GT_ReplyIMsg()** is provided for understanding only; it is crucial that the program make no assumptions or interpretations about the real **IntuiMessage**. Any such inferences are not likely to hold true in the future. See the section on documented side-effects for more information.

IDCMP FLAGS

The various GadTools gadget types require certain classes of IDCMP messages in order to work. Applications specify these IDCMP classes when the window is opened or later with **ModifyIDCMP()** (see the "Intuition Windows" chapter for more on this). Each kind of GadTools gadget requires one or more of these IDCMP classes: IDCMP_GADGETUP, IDCMP_GADGETDOWN, IDCMP_MOUSEMOVE, IDCMP_MOUSEBUTTONS and IDCMP_INTUITICKS. As a convenience, the IDCMP classes required by each kind of gadget are defined in *<libraries/gadtools.h>*. For example, SLIDERIDCMP is defined to be:

```
#define SLIDERIDCMP      (IDCMP_GADGETUP | IDCMP_GADGETDOWN | IDCMP_MOUSEMOVE)
```

> *Always OR the IDCMP Flag Bits.* When specifying the IDCMP classes for a window, never add the flags together, always OR the bits together. Since many of the GadTools IDCMP constants have multiple bits set, adding the values will not lead to the proper flag combination.

If a certain kind of GadTools gadget is used, the window must use all IDCMP classes required by that kind of gadget. Do not omit any that are given for that class, even if the application does require the message type.

Because of the way GadTools gadgets are implemented, programs that use them always require notification about window refresh events. Even if the application performs no rendering of its own, it may not use the WFLG_NOCAREREFRESH window flag and must always set IDCMP_REFRESHWINDOW. See the section on "Gadget Refresh Functions" later in this chapter for more on this.

FREEING GADGETS

After closing the window, the gadgets allocated using **CreateGadget()** must be released. **FreeGadgets()** is a simple call that will free all the GadTools gadgets that it finds, beginning with the gadget whose pointer is passed as an argument.

```
void FreeGadgets( struct Gadget *gad );
```

The **gad** argument is a pointer to the first gadget to be freed. It is safe to call **FreeGadgets()** with a NULL gadget pointer, the function will then return immediately. Before calling **FreeGadgets()**, the application must first either remove the gadgets or close the window.

When the gadget passed to **FreeGadgets()** is the first gadget in a linked list, the function frees all the GadTools gadgets on the list without patching pointers or trying to maintain the integrity of the list. Any non-GadTools gadgets found on the list will not be freed, hence the result will not necessarily form a nice list since any intervening GadTools gadgets will be gone.

See the section on "Creating Gadget Lists" for more information on using linked lists of gadgets.

SIMPLE GADTOOLS GADGET EXAMPLE

The example listed here shows how to use the **NewGadget** structure and the GadTools library functions discussed above to create a simple button gadget.

```
;/* simplegtgadget.c -- execute me to compile me
lc -b1 -cfistq -v -y simplegtgadget
blink FROM LIB:c.o simplegtgadget.o TO simplegtgadget LIB LIB:lc.lib LIB:amiga.lib
quit

Simple example of a GadTools gadget.  Compiled with SAS C v5.10a
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/gadgetclass.h>
#include <libraries/gadtools.h>

#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/gadtools_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

/* Gadget defines of our choosing, to be used as GadgetID's. */
#define MYGAD_BUTTON    (4)

VOID process_window_events(struct Window *);
VOID gadtoolsWindow(VOID);

struct TextAttr Topaz80 = { "topaz.font", 8, 0, 0, };

struct Library *IntuitionBase;
struct Library *GadToolsBase;


/*
** Open all libraries and run.  Clean up when finished or on error..
*/
void main(void)
{
if ( (IntuitionBase = OpenLibrary("intuition.library", 37)) != NULL )
    {
    if ( (GadToolsBase = OpenLibrary("gadtools.library", 37)) != NULL )
        {
        gadtoolsWindow();

        CloseLibrary(GadToolsBase);
        }
    CloseLibrary(IntuitionBase);
    }
}


/*
** Prepare for using GadTools, set up gadgets and open window.
** Clean up and when done or on error.
*/
VOID gadtoolsWindow(VOID)
{
struct Screen    *mysc;
struct Window    *mywin;
struct Gadget    *glist, *gad;
struct NewGadget ng;
void             *vi;

glist = NULL;
```

```
if ( (mysc = LockPubScreen(NULL)) != NULL )
    {
    if ( (vi = GetVisualInfo(mysc, TAG_END)) != NULL )
        {
        /* GadTools gadgets require this step to be taken */
        gad = CreateContext(&glist);

        /* create a button gadget centered below the window title */
        ng.ng_TextAttr   = &Topaz80;
        ng.ng_VisualInfo = vi;
        ng.ng_LeftEdge   = 150;
        ng.ng_TopEdge    = 20 + mysc->WBorTop + (mysc->Font->ta_YSize + 1);
        ng.ng_Width      = 100;
        ng.ng_Height     = 12;
        ng.ng_GadgetText = "Click Here";
        ng.ng_GadgetID   = MYGAD_BUTTON;
        ng.ng_Flags      = 0;
        gad = CreateGadget(BUTTON_KIND, gad, &ng, TAG_END);

        if (gad != NULL)
            {
            if ( (mywin = OpenWindowTags(NULL,
                    WA_Title,     "GadTools Gadget Demo",
                    WA_Gadgets,   glist,      WA_AutoAdjust,    TRUE,
                    WA_Width,       400,      WA_InnerHeight,    100,
                    WA_DragBar,    TRUE,      WA_DepthGadget,   TRUE,
                    WA_Activate,   TRUE,      WA_CloseGadget,   TRUE,
                    WA_IDCMP, IDCMP_CLOSEWINDOW |
                              IDCMP_REFRESHWINDOW | BUTTONIDCMP,
                    WA_PubScreen,   mysc,
                    TAG_END)) != NULL )
                {
                GT_RefreshWindow(mywin, NULL);

                process_window_events(mywin);

                CloseWindow(mywin);
                }
            }
        /* FreeGadgets() must be called after the context has been
        ** created.  It does nothing if glist is NULL
        */
        FreeGadgets(glist);
        FreeVisualInfo(vi);
        }
    UnlockPubScreen(NULL, mysc);
    }
}

/*
** Standard message handling loop with GadTools message handling functions
** used (GT_GetIMsg() and GT_ReplyIMsg()).
*/
VOID process_window_events(struct Window *mywin)
{
struct IntuiMessage *imsg;
struct Gadget *gad;
BOOL  terminated = FALSE;

while (!terminated)
    {
    Wait (1 << mywin->UserPort->mp_SigBit);

    /* Use GT_GetIMsg() and GT_ReplyIMsg() for handling */
    /* IntuiMessages with GadTools gadgets.             */
    while ((!terminated) && (imsg = GT_GetIMsg(mywin->UserPort)))
        {
        /* GT_ReplyIMsg() at end of loop */

        switch (imsg->Class)
            {
            case IDCMP_GADGETUP:       /* Buttons only report GADGETUP */
                gad = (struct Gadget *)imsg->IAddress;
                if (gad->GadgetID == MYGAD_BUTTON)
                        printf("Button was pressed.\n");
                break;
```

```
            case IDCMP_CLOSEWINDOW:
                terminated = TRUE;
                break;
            case IDCMP_REFRESHWINDOW:
                /* This handling is REQUIRED with GadTools. */
                GT_BeginRefresh(mywin);
                GT_EndRefresh(mywin, TRUE);
                break;
            }
        /* Use the toolkit message-replying function here... */
        GT_ReplyIMsg(imsg);
        }
    }
}
```

MODIFYING GADGETS

The attributes of a gadget are set up when the gadget is created. Some of these attributes can be changed later by using the **GT_SetGadgetAttrs()** function:

```
void GT_SetGadgetAttrs (struct Gadget *gad, struct Window *win, struct Requester *req,
                        Tag tag1, ... )
void GT_SetGadgetAttrsA(struct Gadget *gad, struct Window *win, struct Requester *req,
                        struct TagItem *taglist)
```

The **gad** argument specifies the gadget to be changed while the **win** argument specifies the window the gadget is in. Currently, the **req** argument is unused and must be set to NULL.

The gadget attributes are changed by passing tag arguments to these functions. The tag arguments can be either a set of **TagItems** on the stack for **GT_SetGadgetAttrs()**, or a pointer to an array of **TagItems** for **GT_SetGadgetAttrsA()**. The tag items specify the attributes that are to be changed for the gadget. Keep in mind though that not every gadget attribute can be modified this way.

For example, in the slider gadget presented earlier, the level-formatting string may not be changed after the gadget is created. However, the slider's level may be changed to 5 as follows:

```
GT_SetGadgetAttrs(slidergad, win, req,
    GTSL_Level, 5,
    TAG_END);
```

Here are some other example uses of **GT_SetGadgetAttrs()** to change gadget attributes after it is created.

```
/* Disable a button gadget */
GT_SetGadgetAttrs(buttongad, win, NULL,
                        GA_Disabled, TRUE,
                        TAG_END);

/* Change a slider's range to be 1 to 100, currently at 50 */
GT_SetGadgetAttrs(slidergad, win, NULL,
                        GTSL_Min, 1,
                        GTSL_Max, 100,
                        GTSL_Level, 50,
                        TAG_END);

/* Add a node to the head of listview's list, and make it the selected one */
GT_SetGadgetAttrs(listviewgad, win, NULL,
                        /* detach list before modifying */
                        GTLV_Labels, ~0,
                        TAG_END);
AddHead(&lvlabels, &newnode);
GT_SetGadgetAttrs(listviewgad, win, NULL,
                        /* re-attach list */
                        GTLV_Labels, &lvlabels,
                        GTLV_Selected, 0,
                        TAG_END);
```

When changing a gadget using these functions, the gadget will automatically update its visuals. No refresh is required, nor should any refresh call be performed.

> *Warning:* The **GT_SetGadgetAttrs()** functions may not be called inside of a **GT_BeginRefresh()/GT_EndRefresh()** pair. This is true of Intuition gadget functions generally, including those discussed in the "Intuition Gadgets" chapter.

In the sections that follow all the possible attributes for each kind of gadget are discussed. The tags are also described in the Autodocs for **GT_SetGadgetAttrs()** in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*.

> *Important:* Tags that can only be sent to **CreateGadget()** and not to **GT_SetGadgetAttrs()** will be marked as *create only* in the discussion that follows. Those that are valid parameters to both functions will be marked as *create and set*.

THE KINDS OF GADTOOLS GADGETS

This section discusses the unique features of each kind of gadget supported by the GadTools library.

Button Gadgets

Button gadgets (BUTTON_KIND) are perhaps the simplest kind of GadTools gadget. Button gadgets may be used for objects like the "OK" and "Cancel" buttons in requesters. GadTools will create a hit-select button with a raised bevelled border. The label supplied will be centered on the button's face. Since the label is not clipped, be sure that the gadget is large enough to contain the text supplied.

Button gadgets recognize only one tag:

GA_Disabled (BOOL)
: Set this attribute to TRUE to disable or ghost the button gadget, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user selects a button gadget, the program will receive an IDCMP_GADGETUP event.

If clicking on a button causes a requester to appear, for example a button that brings up a color requester, then the button text should end in ellipsis (...), as in "Quit..."

Text-Entry and Number-Entry Gadgets

Text-entry (STRING_KIND) and number-entry (INTEGER_KIND) gadgets are fairly typical Intuition string gadgets. The typing area is contained by a border which is a raised ridge.

Text-entry gadgets accept the following tags:

GTST_String (STRPTR)
: A pointer to the string to be placed into the text-entry gadget buffer or NULL to get an empty text-entry gadget. The string itself is actually copied into the gadget's buffer. The default is NULL. (Create and set.)

GTST_MaxChars (UWORD)
The maximum number of characters that the text-entry gadget should hold. The string buffer that gets created for the gadget will actually be one bigger than this number, in order to hold the trailing NULL. The default is 64. (Create only.)

Number-entry gadgets accept the following tags:

GTIN_Number (ULONG)
The number to be placed into the number-entry gadget. The default is zero. (Create and set.)

GTIN_MaxChars (UWORD)
The maximum number of digits that the number-entry gadget should hold. The string buffer that gets created for the gadget will actually be one bigger than this, in order to hold the trailing NULL. The default is 10. (Create only.)

Both text-entry and number-entry gadgets, which are collectively called string gadgets, accept these common tags:

STRINGA_Justification
This attribute controls the placement of the string or number within its box and can be one of GACT_STRINGLEFT, GACT_STRINGRIGHT or GACT_STRINGCENTER. The default is GACT_STRINGLEFT. (Create only.)

STRINGA_ReplaceMode (BOOL)
Set STRINGA_ReplaceMode to TRUE to get a string gadget which is in replace-mode, as opposed to auto-insert mode. (Create only.)

GA_Disabled (BOOL)
Set this attribute to TRUE to disable the string gadget, otherwise to FALSE. The default is FALSE. (Create and set.)

STRINGA_ExitHelp (BOOL)
(New for V37, ignored under V36). Set this attribute to TRUE if the application wants to hear the Help key from within this string gadget. This feature allows the program to hear the press of the Help key in all cases. If TRUE, pressing the help key while this gadget is active will terminate the gadget and send a message. The program will receive an IDCMP_GADGETUP message having a **Code** value of 0x5F, the rawkey code for Help. Typically, the program will want to reactivate the gadget after performing the help-display. The default is FALSE. (Create only.)

GA_TabCycle (BOOL)
(New for V37, ignored under V36). If the user types Tab or Shift Tab into a GA_TabCycle gadget, Intuition will activate the next or previous such gadget in sequence. This gives the user easy keyboard control over which text-entry or number-entry gadget is active. Tab moves to the next GA_TabCycle gadget in the gadget list and Shift Tab moves to the previous one. When the user presses Tab or Shift Tab, Intuition will deactivate the gadget and send this program an IDCMP_GADGETUP message with the code field set to 0x09, the ASCII value for a tab. Intuition will then activate the next indicated gadget. Check the shift bits of the qualifier field to learn if Shift Tab was typed. The ordering of the gadgets may only be controlled by the order in which they were added to the window. For special cases, for example, if there is only one string gadget in the window, this feature can be suppressed by specifying the tagitem pair **{GA_TabCycle, FALSE}**. The default is TRUE. (Create only.)

GTST_EditHook (struct Hook *)
(New for V37, ignored under V36). Pointer to a custom editing hook for this string or integer gadget. See the ''Intuition Gadgets'' chapter for more information on string gadget edit-hooks.

As with all Intuition string gadgets, the program will receive an IDCMP_GADGETUP message only when the user presses Enter, Return, Help, Tab or Shift Tab while typing in the gadget. Note that, like Intuition string gadgets, the program will not hear anything if the user deactivates the string gadget by clicking elsewhere. Therefore, it is a good idea to always check the string gadget's buffer before using its contents, instead of just tracking its value as IDCMP_GADGETUP messages are received for this gadget.

Be sure the code is designed so that nothing drastic happens, like closing a requester or opening a file, if the IDCMP_GADGETUP message has a non-zero **Code** field; the program will want to handle the Tab and Help cases intelligently.

To read the string gadget's buffer, look at the **Gadget**'s **StringInfo Buffer**:

```
((struct StringInfo *)gad->SpecialInfo)->Buffer
```

To determine the value of an integer gadget, look at the **Gadget**'s **StringInfo LongInt** in the same way.

Always use the GTST_String or GTIN_Number tags to set these values. Never write to the **StringInfo->Buffer** or **StringInfo->LongInt** fields directly.

GadTools string and integer gadgets do not directly support the GA_Immediate property (which would cause Intuition to send an IDCMP_GADGETDOWN event when such a gadget is first selected). However, this property can be very important. Therefore, the following technique can be used to enable this property.

> *Warning:* Note that the technique shown here relies on directly setting flags in a GadTools gadget; this is not normally allowed since it hinders future compatibility. Do not attempt to change other flags or properties of GadTools gadgets except through the defined interfaces of **CreateGadgetA()** and **GT_SetGadgetAttrsA()**. Directly modifying flags or properties is legal only when officially sanctioned by Commodore.

To get the GA_Immediate property, pass the **{GA_Immediate,TRUE}** tag to **CreateGadgetA()**. Even though this tag is ignored for string and integer gadgets under V37, this will allow future versions of GadTools to learn of your request in the correct way. Then, under V37 only, set the GACT_IMMEDIATE flag in the gadget's **Activation** field:

```
gad = CreateGadget( STRING_KIND, gad, &ng,
      /* string gadget tags go here */
      GTST_...,

      /* Add this tag for future GadTools releases */
      GA_Immediate, TRUE,
      ...
      TAG_DONE );

if ( ( gad ) && ( GadToolsBase->lib_Version == 37) )
{
      /* Under GadTools V37 only, set this attribute
       * directly.  Do not set this attribute under
       * future versions of GadTools, or for gadgets
       * other than STRING_KIND or INTEGER_KIND.
       */
      gad->Activation |= GACT_IMMEDIATE;
}
```

Checkbox Gadgets

Checkboxes (CHECKBOX_KIND) are appropriate for presenting options which may be turned on or off. This kind of gadget consists of a raised box which contains a checkmark if the option is selected or is blank if the option is not selected. Clicking on the box toggles the state of the checkbox.

The width and height of a checkbox are currently fixed (to 26x11). If variable-sized checkboxes are added in the future, they will be done in a compatible manner. Currently the width and height specified in the **NewGadget** structure are ignored.

The checkbox may be controlled with the following tags:

GTCB_Checked (BOOL)
: Set this attribute to TRUE to set the gadget's state to *checked.* Set it to FALSE to mark the gadget as *unchecked.* The default is FALSE. (Create and set.)

GA_Disabled (BOOL)
: Set this attribute to TRUE to disable the checkbox, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user selects a checkbox, the program will receive an **IntuiMessage** with a class of IDCMP_GADGETUP. As this gadget always toggles, the program can easily track the state of the gadget. Feel free to read the **gadget->Flags** GFLG_SELECTED bit. Note, however, that the **Gadget** structure itself is not synchronized to the **IntuiMessages** received. If the user clicks a second time, the GFLG_SELECTED bit can toggle again before the program gets a chance to read it. This is true of any of the dynamic fields of the **Gadget** structure, and is worth being aware of, although only rarely will an application have to account for it.

Mutually-Exclusive Gadgets

Use mutually exclusive gadgets (MX_KIND), or *radio buttons*, when the user must choose only one option from a short list of possibilities. Mutually exclusive gadgets are appropriate when there are a small number of choices, perhaps eight or less.

A set of mutually exclusive gadgets consists of a list of labels and beside each label, a small raised oval that looks like a button. Exactly one of the ovals is recessed and highlighted, to indicate the selected choice. The user can pick another choice by clicking on any of the raised ovals. This choice will become active and the previously selected choice will become inactive. That is, the selected oval will become recessed while the previous one will pop out, like the buttons on a car radio.

Mutually exclusive gadgets recognize these tags:

GTMX_Labels (STRPTR *)
: A NULL-pointer-terminated array of strings which are to be the labels beside each choice in the set of mutually exclusive gadgets. This array determines how many buttons are created. This array must be supplied to **CreateGadget()** and may not be changed. The strings themselves must remain valid for the lifetime of the gadget. (Create only.)

GTMX_Active (UWORD)
: The ordinal number, counting from zero, of the active choice of the set of mutually exclusive gadgets. The default is zero. (Create and set.)

GTMX_Spacing (UWORD)
The amount of space, in pixels, that will be placed between successive choices in a set of mutually exclusive gadgets. The default is one. (Create only.)

When the user selects a new choice from a set of mutually exclusive gadgets, the program will receive an IDCMP_GADGETDOWN **IntuiMessage**. Look in the **IntuiMessage**'s **Code** field for the ordinal number of the new active selection.

The **ng_GadgetText** field of the **NewGadget** structure is ignored for mutually exclusive gadgets. The text position specified in **ng_Flags** determines whether the item labels are placed to the left or the right of the radio buttons themselves. By default, the labels appear on the left. Do not specify PLACETEXT_ABOVE, PLACETEXT_BELOW or PLACETEXT_IN for this kind of gadget.

Like the checkbox, the size of the radio button is currently fixed, and the dimensions supplied in the **NewGadget** structure are ignored. If in the future the buttons are made scalable, it will be done in a compatible manner. Currently, mutually exclusive gadgets may not be disabled.

Cycle Gadgets

Like mutually exclusive gadgets, cycle gadgets (CYCLE_KIND) allow the user to choose exactly one option from among several.

The cycle gadget appears as a raised rectangular button with a vertical divider near the left side. A circular arrow glyph appears to the left of the divider, while the current choice appears to the right. Clicking on the cycle gadget advances to the next choice, while shift-clicking on it changes it to the previous choice.

Cycle gadgets are more compact than mutually exclusive gadgets, since only the current choice is displayed. They are preferable to mutually exclusive gadgets when a window needs to have several such gadgets as in the PrinterGfx Preferences editor, or when there is a medium number of choices. If the number of choices is much more than about a dozen, it may become too frustrating and inefficient for the user to find the desired choice. In that case, use a listview (scrolling list) instead.

The tags recognized by cycle gadgets are:

GTCY_Labels (STRPTR *)
Like GTMX_Labels, this tag is associated with a NULL-pointer-terminated array of strings which are the choices that this gadget allows. This array must be supplied to **CreateGadget()**, and can only be changed starting in V37. The strings themselves must remain valid for the lifetime of the gadget. (Create only (V36), Create and set (V37).)

GTCY_Active (UWORD)
The ordinal number, counting from zero, of the active choice of the cycle gadget. The default is zero. (Create and set.)

GA_Disabled (BOOL)
(New for V37, ignored by V36.) Set this attribute to TRUE to disable the cycle gadget, to FALSE otherwise. The default is FALSE. (Create and set.)

When the user clicks or shift-clicks on a cycle gadget, the program will receive an IDCMP_GADGETUP **IntuiMessage**. Look in the **Code** field of the **IntuiMessage** for the ordinal number of the new active selection.

Slider Gadgets

Sliders are one of the two kinds of proportional gadgets offered by GadTools. Slider gadgets (SLIDER_KIND) are used to control an amount, a level or an intensity, such as volume or color. Scroller gadgets (SCROLLER_KIND) are discussed below.

Slider gadgets accept the following tags:

GTSL_Min (WORD)
The minimum level of a slider. The default is zero. (Create and set.)

GTSL_Max (WORD)
The maximum level of a slider. The default is 15. (Create and set.)

GTSL_Level (WORD)
The current level of a slider. The default is zero. When the level is at its minimum, the knob will be all the way to the left for a horizontal slider or all the way at the bottom for a vertical slider. Conversely, the maximum level corresponds to the knob being to the extreme right or top. (Create and set.)

GTSL_LevelFormat (STRPTR)
The current level of the slider may be displayed in real-time alongside the gadget. To use the level-display feature, the program must be using a monospace font for this gadget.

GTSL_LevelFormat specifies a **printf()**-style formatting string used to render the slider level beside the slider (the complete set of formatting options is described in the Exec library function **RawDoFmt()**). Be sure to use the 'l' (long word) modifier for the number. Field-width specifiers may be used to ensure that the resulting string is always of constant width. The simplest would be "%2ld". A 2-digit hexadecimal slider might use "%02lx", which adds leading zeros to the number. Strings with extra text, such as "%3ld hours", are permissible. If this tag is specified, the program must also provide **GTSL_MaxLevelLen**. By default, the level is not displayed. (Create only.)

GTSL_MaxLevelLen (UWORD)
The maximum length of the string that will result from the given level-formatting string. If this tag is specified, the program must also provide **GTSL_LevelFormat**. By default, the level is not displayed. (Create only.)

GTSL_LevelPlace
To choose where the optional display of the level is positioned. It must be one of PLACETEXT_LEFT, PLACETEXT_RIGHT, PLACETEXT_ABOVE or PLACETEXT_BELOW. The level may be placed anywhere with the following exception: the level and the label may not be both above or both below the gadget. To place them both on the same side, allow space in the gadget's label (see the example). The default is PLACETEXT_LEFT. (Create only.)

GTSL_DispFunc (LONG (*function)(struct Gadget *, WORD))

Optional function to convert the level for display. A slider to select the number of colors for a screen may operate in screen depth (1 to 5, for instance), but actually display the number of colors (2, 4, 8, 16 or 32). This may be done by providing a **GTSL_DispFunc** function which returns **1 << level**. The function must take a pointer to the **Gadget** as the first parameter and the level, a WORD, as the second and return the result as a LONG. The default behavior for displaying a level is to do so without any conversion. (Create only.)

GA_Immediate (BOOL)

Set this to TRUE to receive an IDCMP_GADGETDOWN **IntuiMessage** when the user presses the mouse button over the slider. The default is FALSE. (Create only.)

GA_RelVerify (BOOL)

Set this to TRUE to receive an IDCMP_GADGETUP **IntuiMessage** when the user releases the mouse button after using the slider. The default is FALSE. (Create only.)

PGA_Freedom

Specifies which direction the knob may move. Set to LORIENT_VERT for a vertical slider or LORIENT_HORIZ for a horizontal slider. The default is LORIENT_HORIZ. (Create only.)

GA_Disabled (BOOL)

Set this attribute to TRUE to disable the slider, to FALSE otherwise. The default is FALSE. (Create and set.)

Up to three different classes of **IntuiMessage** may be received at the port when the user plays with a slider, these are IDCMP_MOUSEMOVE, IDCMP_GADGETUP and IDCMP_GADGETDOWN. The program may examine the **IntuiMessage Code** field to discover the slider's level.

IDCMP_MOUSEMOVE **IntuiMessages** will be heard whenever the slider's level changes. IDCMP_MOUSEMOVE **IntuiMessages** will not be heard if the knob has not moved far enough for the level to actually change. For example if the slider runs from 0 to 15 and is currently set to 12, if the user drags the slider all the way up the program will hear no more than three IDCMP_MOUSEMOVEs, one each for 13, 14 and 15.

If {**GA_Immediate, TRUE**} is specified, then the program will always hear an IDCMP_GADGETDOWN **IntuiMessage** when the user begins to adjust a slider. If {**GA_RelVerify, TRUE**} is specified, then the program will always hear an IDCMP_GADGETUP **IntuiMessage** when the user finishes adjusting the slider. If IDCMP_GADGETUP or IDCMP_GADGETDOWN **IntuiMessages** are requested, the program will always hear them, even if the level has not changed since the previous **IntuiMessage**.

Note that the **Code** field of the **IntuiMessage** structure is a UWORD, while the slider's level may be negative, since it is a WORD. Be sure to copy or cast the **IntuiMessage->Code** into a WORD if the slider has negative levels.

If the user clicks in the container next to the knob, the slider level will increase or decrease by one. If the user drags the knob itself, then the knob will snap to the nearest integral position when it is released.

Here is an example of the screen-depth slider discussed earlier:

```
/* NewGadget initialized here. Note the three spaces
 * after "Slider:", to allow a blank plus the two digits
 * of the level display
 */
ng.ng_Flags = PLACETEXT_LEFT;
ng.ng_GadgetText = "Slider:   ";

LONG DepthToColors(struct Gadget *gad, WORD level)
{
return ((WORD)(1 << level));
}


gad = CreateGadget(SLIDER_KIND, gad, &ng,
    GTSL_Min, 1,
    GTSL_Max, 5,
    GTSL_Level, current_depth,
    GTSL_MaxLevelLen, 2,
    GTSL_LevelFormat, "%2ld",
    GTSL_DispFunc, DepthToColors,
    TAG_END);
```

Scroller Gadgets

Scrollers (SCROLLER_KIND) bear some similarity to sliders, but are used for a quite different job: they allow the user to adjust the position of a limited view into a larger area. For example, Workbench's windows have scrollers that allow the user to see icons that are outside the visible portion of a window. Another example is a scrolling list in a file requester which has a scroller that allows the user to see different parts of the whole list.

A scroller consists of a proportional gadget and usually also has a pair of arrow buttons.

While the slider deals in minimum, maximum and current level, the scroller understands **Total**, **Visible** and **Top**. For a scrolling list, **Total** would be the number of items in the entire list, **Visible** would be the number of lines visible in the display area and **Top** would be the number of the first line displayed in the visible part of the list. **Top** would run from zero to **Total - Visible**. For an area-scroller such as those in Workbench's windows, **Total** would be the height (or width) of the whole area, **Visible** would be the visible height (or width) and **Top** would be the top (or left) edge of the visible part.

Note that the position of a scroller should always represent the position of the visible part of the project and never the position of a cursor or insertion point.

Scrollers respect the following tags:

GTSC_Top (WORD)
: The top line or position visible in the area that the scroller represents. The default is zero. (Create and set.)

GTSC_Total (WORD)
: The total number of lines or positions that the scroller represents. The default is zero. (Create and set.)

GTSC_Visible (WORD)
: The visible number of lines or positions that the scroller represents. The default is two. (Create and set.)

GTSC_Arrows (UWORD)
Asks for arrow gadgets to be attached to the scroller. The value supplied will be used as the width of each arrow button for a horizontal scroller or the height of each arrow button for a vertical scroller, the other dimension will be set by GadTools to match the scroller size. It is generally recommend that arrows be provided. The default is no arrows. (Create only.)

GA_Immediate (BOOL)
Set this to TRUE to receive an IDCMP_GADGETDOWN **IntuiMessage** when the user presses the mouse button over the scroller. The default is FALSE. (Create only.)

GA_RelVerify (BOOL)
Set this to TRUE to receive an IDCMP_GADGETUP **IntuiMessage** when the user releases the mouse button after using the scroller. The default is FALSE. (Create only.)

PGA_Freedom
Specifies which direction the knob may move. Set to LORIENT_VERT for a vertical scroller or LORIENT_HORIZ for a horizontal scroller. The default is LORIENT_HORIZ. (Create only.)

GA_Disabled (BOOL)
Set this attribute to TRUE to disable the scroller, to FALSE otherwise. The default is FALSE. (Create and set.)

The **IntuiMessages** received for a scroller gadget are the same in nature as those for a slider defined above, including the fact that messages are only heard by the program when the knob moves far enough for the **Top** value to actually change. The **Code** field of the **IntuiMessage** will contain the new **Top** value of the scroller.

If the user clicks on an arrow gadget, the scroller moves by one unit. If the user holds the button down over an arrow gadget, it repeats.

If the user clicks in the container next to the knob, the scroller will move by one page, which is the visible amount less one. This means that when the user pages through a scrolling list, any pair of successive views will overlap by one line. This helps the user understand the continuity of the list. If the program is using a scroller to pan through an area then there will be an overlap of one unit between successive views. It is recommended that **Top**, **Visible** and **Total** be scaled so that one unit represents about five to ten percent of the visible amount.

Listview Gadgets

Listview gadgets (LISTVIEW_KIND) are scrolling lists. They consist of a scroller with arrows, an area where the list itself is visible and optionally a place where the current selection is displayed, which may be editable. The user can browse through the list using the scroller or its arrows and may select an entry by clicking on that item.

There are a number of tags that are used with listviews:

GTLV_Labels (struct List *)
An Exec list whose nodes' **ln_Name** fields are to be displayed as items in the scrolling list. If the list is empty, an empty **List** structure or a NULL value may be used for **GTLV_Labels**. This tag accepts a value of "~0" to detach the list from the listview, defined below. The default is NULL. (Create and set.)

GTLV_Top (UWORD)
The ordinal number of the top item visible in the listview. The default is zero. (Create and set.)

GTLV_ReadOnly (BOOL)
Set this to TRUE for a read-only listview, which the user can browse, but not select items from. A read-only listview can be recognized because the list area is recessed, not raised. The default is FALSE. (Create only.)

GTLV_ScrollWidth (UWORD)
The width of the scroller to be used in the listview. Any value specified must be reasonably bigger than zero. The default is 16. (Create only.)

GTLV_ShowSelected (struct Gadget *)
Use this tag to show the currently selected entry displayed underneath the listview. Set its value to NULL to get a read-only (TEXT_KIND) display of the currently selected entry or set it to a pointer to an already-created GadTools STRING_KIND gadget to allow the user to directly edit the current entry. By default, there is no display of the currently selected entry. (Create only.)

GTLV_Selected (UWORD)
Ordinal number of the item to be placed into the display of the current selection under the listview. This tag is ignored if **GTLV_ShowSelected** is not used. Set it to "~0" to have no current selection. The default is "~0". (Create and set.)

LAYOUTA_Spacing (UWORD)
Extra space, in pixels, to be placed between the entries in the listview. The default is zero. (Create only.)

The program will only hear from a listview when the user selects an item from the list. The program will then receive an IDCMP_GADGETUP **IntuiMessage**. This message will contain the ordinal number of the item within the list that was selected in the **Code** field of the message. This number is independent of the displayed listview, it is the offset from the start of the list of items.

If the program attaches a display gadget by using the **TagItem {GTLV_ShowSelected, NULL}**, then whenever the user clicks on an entry in the listview it will be copied into the display gadget.

If the display gadget is to be editable, then the program must first create a GadTools STRING_KIND gadget whose width matches the width of the listview. The **TagItem {GTLV_ShowSelected, stringgad}** is used to install the editable gadget, where **stringgad** is the pointer returned by **CreateGadget()**. When the user selects any entry from the listview, it gets copied into the string gadget. The user can edit the string and the program will hear normal string gadget IDCMP_GADGETUP messages from the STRING_KIND gadget.

The Exec **List** and its **Node** structures may not be modified while they are attached to the listview, since the list might be needed at any time. If the program has prepared an entire new list, including a new **List** structure and all new nodes, it may replace the currently displayed list in a single step by calling **GT_SetGadgetAttrs()** with the **TagItem {GTLV_Labels, newlist}**. If the program needs to operate on the list that has already been passed to the listview, it should detach the list by setting the **GTLV_Labels** attribute to "~0". When done modifying the list, resubmit it by setting **GTLV_Labels** to once again point to it. This is better than first setting the labels to NULL and later back to the list, since setting **GTLV_Labels** to NULL will visually clear the listview. If the **GTLV_Labels** attribute is set to "~0", the program is expected to set it back to something determinate, either a list or NULL, soon after.

The height specified for the listview will determine the number of lines in the list area. When creating a listview, it will be no bigger than the size specified in the **NewGadget** structure. The size will include the current-display gadget, if any, that has been requested via the **GTLV_ShowSelected** tag. The listview may end up being less tall than the application asked for, since the calculated height assumes an integral number of lines in the list area.

By default, the gadget label will be placed above the listview. This may be overridden using **ng_Flags**.

Currently, a listview may not be disabled.

Palette Gadgets

Palette gadgets (PALETTE_KIND) let the user pick a color from a set of several. A palette gadget consists of a number of colored squares, one for each color available. There may also be an optional indicator square which is filled with the currently selected color. To create a color editor, a palette gadget would be combined with some sliders to control red, green and blue components, for example.

Palette gadgets use the following tags:

GTPA_Depth (UWORD)
The number of bitplanes that the palette represents. There will be **1 << depth** squares in the palette gadget. The default is one. (Create only.)

GTPA_Color (UBYTE)
The selected color of the palette. The default is one. (Create and set.)

GTPA_ColorOffset (UBYTE)
The first color to use in the palette. For example, if **GTPA_Depth** is two and **GTPA_ColorOffset** is four, then the palette will have squares for colors four, five, six and seven. The default is zero. (Create only.)

GTPA_IndicatorWidth (UWORD)
The desired width of the current-color indicator. By specifying this tag, the application is asking for an indicator to be placed to the left of the color selection squares. The indicator will be as tall as the gadget itself. By default there is no indicator. (Create only.)

GTPA_IndicatorHeight (UWORD)
The desired height of the current-color indicator. By specifying this tag, the application is asking for an indicator to be placed above the color selection squares. The indicator will be as wide as the gadget itself. By default there is no indicator. (Create only.)

GA_Disabled (BOOL)
Set this attribute to TRUE to disable the palette gadget, to FALSE otherwise. The default is FALSE. (Create and set.)

An IDCMP_GADGETUP **IntuiMessage** will be received when the user selects a color from the palette. The current-color indicator is recessed, indicating that clicking on it has no effect.

If the palette is wide and not tall, use the **GTPA_IndicatorWidth** tag to put the indicator on the left. If the palette is tall and narrow, put the indicator on top using **GTPA_IndicatorHeight**.

By default, the gadget's label will go above the palette gadget, unless **GTPA_IndicatorWidth** is specified, in which case the label will go on the left. In either case, the default may be overridden by setting the appropriate flag in the **NewGadget**'s **ng_Flags** field.

The size specified for the palette gadget will determine how the area is subdivided to make the individual color squares. The actual size of the palette gadget will be no bigger than the size given, but it can be smaller in order to make the color squares all exactly the same size.

Text-Display and Numeric-Display Gadgets

Text-display (TEXT_KIND) and numeric-display (NUMBER_KIND) gadgets are read-only displays of information. They are useful for displaying information that is not editable or selectable, while allowing the application to use the GadTools formatting and visuals. Conveniently, the visuals are automatically refreshed through normal GadTools gadget processing. The values displayed may be modified by the program in the same way other GadTools gadgets may be updated.

Text-display and number-display gadgets consist of a fixed label (the one supplied as the **NewGadget**'s **ng_GadgetText**), as well as a changeable string or number (**GTTX_Text** or **GTNM_Number** respectively). The fixed label is placed according to the PLACETEXT_ flag chosen in the **NewGadget** **ng_Flags** field. The variable part is aligned to the left-edge of the gadget.

Text-display gadgets recognize the following tags:

GTTX_Text (STRPTR)
Pointer to the string to be displayed or NULL for no string. The default is NULL. (Create and set.)

GTTX_Border (BOOL)
Set to TRUE to place a recessed border around the displayed string. The default is FALSE. (Create only.)

GTTX_CopyText (BOOL)
This flag instructs the text-display gadget to copy the supplied **GTTX_Text** string instead of using only a pointer to the string. This only works for the value of **GTTX_Text** set at **CreateGadget()** time. If **GTTX_Text** is changed, the new text will be referenced by pointer, not copied. Do not use this tag without a non-NULL **GTTX_Text**. (Create only.)

Number-display gadgets have the following tags:

GTNM_Number (LONG)
The number to be displayed. The default is zero. (Create or set.)

GTNM_Border (BOOL)
Set to TRUE to place a recessed border around the displayed number. The default is FALSE. (Create only.)

Since they are not selectable, text-display and numeric-display gadgets never cause **IntuiMessages** to be sent to the application.

Generic Gadgets

If the application requires a specialized gadget which does not fit into any of the defined GadTools kinds but would still like to use the GadTools gadget creation and deletion functions, it may create a GadTools generic gadget and use it any way it sees fit. In fact, all of the kinds of GadTools gadgets are created out of GadTools GENERIC_KIND gadgets.

The gadget that gets created will heed almost all the information contained in the **NewGadget** structure supplied.

If **ng_GadgetText** is supplied, the gadget's **GadgetText** will point to an **IntuiText** structure with the provided string and font. However, do not specify any of the PLACETEXT **ng_Flags**, as they are currently ignored by GENERIC_KIND gadgets. PLACETEXT flags may be supported by generic GadTools gadgets in the future.

It is up to the program to set the **Flags**, **Activation**, **GadgetRender**, **SelectRender**, **MutualExclude** and **SpecialInfo** fields of the **Gadget** structure.

The application must also set the **GadgetType** field, but be certain to preserve the bits set by **CreateGadget()**. For instance, to make a gadget boolean, use:

```
gad->GadgetType |= GTYP_BOOLGADGET;
```

and not

```
gad->GadgetType = GTYP_BOOLGADGET;
```

Using direct assignment, (the = operator), clears all other flags in the **GadgetType** field and the gadget may not be properly freed by **FreeGadgets()**.

FUNCTIONS FOR SETTING UP GADTOOLS MENUS AND GADGETS

This section gives all the details on the functions used to set up GadTools menus and gadgets that were mentioned briefly earlier in this chapter.

GetVisualInfo() and FreeVisualInfo()

In order to ensure their best appearance, GadTools gadgets and menus need information about the screen on which they will appear. Before creating any GadTools gadgets or menus, the program must get this information using the **GetVisualInfo()** call.

```
APTR GetVisualInfoA( struct Screen *screen, struct TagItem *taglist );
APTR GetVisualInfo( struct Screen *screen, Tag tag1, ... );
```

Set the **screen** argument to a pointer to the screen you are using. The tag arguments, **tag1** or **taglist**, are reserved for future extensions. Currently none are recognized, so only TAG_END should be used.

The function returns an abstract handle called the **VisualInfo**. For GadTools gadgets, the **ng_VisualInfo** field of the **NewGadget** structure must be set to this handle before the gadget can be added to the window. GadTools menu layout and creation functions also require the **VisualInfo** handle as an argument.

There are several ways to get the pointer to the screen on which the window will be opened. If the

application has its own custom screen, this pointer is known from the call to **OpenScreen()** or **OpenScreenTags()**. If the application already has its window opened on the Workbench or some other public screen, the screen pointer can be found in **Window.WScreen**. Often the program will create its gadgets and menus before opening the window. In this case, use **LockPubScreen()** to get a pointer to the desired public screen, which also provides a lock on the screen to prevent it from closing. See the chapters "Intuition Screens" and "Intuition Windows" for more about public screens.

The **VisualInfo** data must be freed after all the gadgets and menus have been freed but before releasing the screen. Custom screens are released by calling **CloseScreen()**, public screens are released by calling **CloseWindow()** or **UnlockPubScreen()**, depending on the technique used. Use **FreeVisualInfo()** to free the visual info data.

```
void FreeVisualInfo( APTR vi );
```

This function takes just one argument, the **VisualInfo** handle as returned by **GetVisualInfo()**. The sequence of events for using the **VisualInfo** handle could look like this:

```
init()
{
myscreen = LockPubScreen(NULL);
if (!myscreen)
    {
    cleanup("Failed to lock default public screen");
    }
vi = GetVisualInfo(myscreen);
if (!vi)
    {
    cleanup("Failed to GetVisualInfo");
    }
/* Create gadgets here */
ng.ng_VisualInfo = vi;
}

void cleanup(STRPTR errorstr)
{
/* These functions may be safely called with a NULL parameter: */
FreeGadgets(glist);
FreeVisualInfo(vi);

if (myscreen)
    UnlockPubScreen(NULL, myscreen);

printf(errorstr);
}
```

CreateContext()

Use of GadTools gadgets requires some per-window context information. **CreateContext()** establishes a place for that information to go. This function must be called before any GadTools gadgets are created.

```
struct Gadget *CreateContext( struct Gadget **glistptr );
```

The **glistptr** argument is a double-pointer to a **Gadget** structure. More specifically, this is a pointer to a NULL-initialized pointer to a **Gadget** structure.

The return value of **CreateContext()** is a pointer to this gadget, which should be fed to the program's first call to **CreateGadget()**. This pointer to the **Gadget** structure returned by **CreateContext()**, may then serve as a handle to the list of gadgets as they are created. The code fragment listed in the next section shows how to use **CreateContext()** together with **CreateGadget()** to make a linked list of GadTools gadgets.

CREATING GADGET LISTS

In the discussion of **CreateGadget()** presented earlier, the examples showed only how to make a single gadget. For most applications that use GadTools, however, a whole list of gadgets will be needed. To do this, the application could use code such as this:

```
struct NewGadget *newgad1, *newgad2, *newgad3;
struct Gadget *glist = NULL;
struct Gadget *pgad;

/* Initialize NewGadget structures */

/* Note that CreateContext() requires a POINTER to a NULL-initialized
 * pointer to struct Gadget:
 */
pgad = CreateContext(&glist);

pgad = CreateGadget(BUTTON_KIND, pgad, newgad1, TAG_END);
pgad = CreateGadget(STRING_KIND, pgad, newgad2, TAG_END);
pgad = CreateGadget(MX_KIND,     pgad, newgad3, TAG_END);

if (!pgad)
    {
    FreeGadgets(glist);
    exit_error();
    }
else
    {
    if ( mywin=OpenWindowTags(NULL,
                              WA_Gadgets,   glist,
                              ...
                              /* Other tags... */
                              ...
                              TAG_END) )
        {
        /* Complete the rendering of the gadgets */
        GT_RefreshWindow(win, NULL);
        ...
        /* and continue on... */
        ...
        CloseWindow(mywin);
        }

    FreeGadgets(glist);
    }
```

The pointer to the previous gadget, **pgad** in the code fragment above, is used for three purposes. First, when **CreateGadget()** is called multiple times, each new gadget is automatically linked to the previous gadget's **NextGadget** field, thus creating a gadget list. Second, if one of the gadget creations fails (usually due to low memory, but other causes are possible), then for the next call to **CreateGadget()**, **pgad** will be NULL and **CreateGadget()** will fail immediately. This means that the program can perform several successive calls to **CreateGadget()** and only have to check for failure at the end.

Finally, although this information is hidden in the implementation and not important to the application, certain calls to **CreateGadget()** actually cause several Intuition gadgets to be allocated and these are automatically linked together without program interaction, but only if a previous gadget pointer is supplied. If several gadgets are created by a single **CreateGadget()** call, they work together to provide the functionality of a single GadTools gadget. The application should always act as though the gadget pointer returned by **CreateGadget()** points to a single gadget instance. See ''Documented Side-Effects'' for a warning.

There is one exception to the fact that a program only has to check for failure after the last **CreateGadget()** call and that is when the application depends on the successful creation of a gadget and caches or immediately uses the gadget pointer returned by **CreateGadget()**.

For instance, if the program wants to create a string gadget and save a pointer to the string buffer, it might do so as follows:

```
gad = CreateGadget(STRING_KIND, gad, &ng,
                        GTST_String, "Hello World",
                        TAG_END);
if (gad)
    {
    stringbuffer = ((struct StringInfo *)(gad->SpecialInfo))->Buffer;
    }

/* Creation can continue here: */
gad = CreateGadget(..._KIND, gad, &ng2,
    ...
    TAG_END);
```

A major benefit of having a reusable **NewGadget** structure is that often many fields do not change and some fields change incrementally. For example, the application can set just the **NewGadget**'s **ng_VisualInfo** and **ng_TextAttr** only once and never have to modify them again even if the structure is reused to create many gadgets. A set of similar gadgets may share size and some positional information so that code such as the following might be used:

```
/* Assume that the NewGadget structure 'ng' is fully
 * initialized here for a button labelled "OK"
 */
gad = CreateGadget(BUTTON_KIND, gad, &ng,
    TAG_END);

/* Modify only those fields that need to change: */
ng.ng_GadgetID++;
ng.ng_LeftEdge += 80;
ng.ng_GadgetText = "Cancel";
gad = CreateGadget(BUTTON_KIND, gad, &ng,
                        TAG_END);
```

> *Warning:* All gadgets created by GadTools currently have the GADTOOL_TYPE bit set in their **GadgetType** field. It is not correct to check for, set, clear or otherwise rely on this since it is subject to change.

GADGET REFRESH FUNCTIONS

Normally, GadTools gadgets are created and then attached to a window when the window is opened, either through the WA_FirstGadget tag or the **NewWindow.FirstGadget** field. Alternately, they may be added to a window after it is open by using the functions **AddGList()** and **RefreshGList()**.

Regardless of which way gadgets are attached to a window, the program must then call the **GT_RefreshWindow()** function to complete the rendering of GadTools gadgets. This function takes two arguments.

```
void GT_RefreshWindow( struct Window *win, struct Requester *req );
```

This **win** argument is a pointer to the window that contains the GadTools gadgets. The **req** argument is currently unused and should be set to NULL. This function should only be called immediately after adding GadTools gadgets to a window. Subsequent changes to GadTools gadget imagery made through calls to **GT_SetGadgetAttrs()** will be automatically performed by GadTools when the changes are made. (There is no need to call **GT_RefreshWindow()** in that case.)

As mentioned earlier, applications must always ask for notification of window refresh events for any window that uses GadTools gadgets. When the application receives an IDCMP_REFRESHWINDOW message for a window, Intuition has already refreshed its gadgets. Normally, a program would then call Intuition's **BeginRefresh()**, perform its own custom rendering operations, and finally call **EndRefresh()**. But for a window that uses GadTools gadgets, the application must call **GT_BeginRefresh()** and **GT_EndRefresh()** in place of **BeginRefresh()** and **EndRefresh()**. This allows the the GadTools gadgets to be fully refreshed.

```
void GT_BeginRefresh( struct Window *win );
void GT_EndRefresh ( struct Window *win, long complete );
```

For both functions, the **win** argument is a pointer to the window to be refreshed. For **GT_EndRefresh()**, set the **complete** argument to TRUE if refreshing is complete, set it to FALSE otherwise. See the discussion of **BeginRefresh()** and **EndRefresh()** in the "Intuition Windows" chapter for more about window refreshing.

When using GadTools gadgets, the program may not set the window's WFLG_NOCAREREFRESH flag. Even if there is no custom rendering to be performed, GadTools gadgets requires this minimum code to handle IDCMP_REFRESHWINDOW messages:

```
case IDCMP_REFRESHWINDOW:
    GT_BeginRefresh(win);
    /* custom rendering, if any, goes here */
    GT_EndRefresh(win, TRUE);
    break;
```

OTHER GADTOOLS FUNCTIONS

This section discusses some additional support functions in the GadTools library that serve special needs.

GT_FilterIMsg() and GT_PostFilterIMsg()

For most GadTools programs, **GT_GetIMsg()** and **GT_ReplyIMsg()** work perfectly well. In rare cases an application may find they pose a bit of a problem. A typical case is when all messages are supposed to go through a centralized **ReplyMsg()** that cannot be converted to a **GT_ReplyIMsg()**. Since calls to **GT_GetIMsg()** and **GT_ReplyIMsg()** must be paired, there would be a problem.

For such cases, the **GT_FilterIMsg()** and **GT_PostFilterIMsg()** functions are available. These functions allow **GetMsg()** and **ReplyMsg()** to be used in a way that is compatible with GadTools.

> *Warning:* These functions are for specialized use only and will not be used by the majority of applications. See **GT_GetIMsg()** and **GT_ReplyIMsg()** for standard message handling.

```
struct IntuiMessage *GT_FilterIMsg( struct IntuiMessage *imsg );
struct IntuiMessage *GT_PostFilterIMsg( struct IntuiMessage *imsg );
```

The **GT_FilterIMsg()** function should be called right after **GetMsg()**. It takes a pointer to the original **IntuiMessage** and, if the message applies to a GadTools gadget, returns either a modified **IntuiMessage** or a NULL. A NULL return signifies that the message was consumed by a GadTools gadget (and not needed by the application).

The **GT_PostFilterIMsg()** function should be called before replying to any message modified by **GT_FilterIMsg()**. It takes a pointer to the modified version of an **IntuiMessage** obtained with **GT_FilterIMsg()** and returns a pointer to the original **IntuiMessage**.

The typical calling sequence for a program that uses these functions, is to call **GetMsg()** to get the **IntuiMessage**. Then, if the message applies to a window which contains GadTools gadgets, call **GT_FilterIMsg()**. Any message returned by **GT_FilterIMsg()** should be used like a message returned from **GT_GetIMsg()**.

When done with the message, the application must call **GT_PostFilterIMsg()** to perform any clean up necessitated by the previous call to **GT_FilterIMsg()**. In all cases, the application *must* then reply the original **IntuiMessage** using **ReplyMsg()**. This is true even for consumed messages as these are *not* replied by GadTools. For example, the application could use code such as this:

```
/* port is a message port receiving different messages */
/* gtwindow is the window that contains GadTools gadgets */

imsg = GetMsg(port);

/* Is this the window with GadTools gadgets? */
if (imsg->IDCMPWindow == gtwindow)
    {
    /* Filter the message and see if action is needed */
    if (gtimsg = GT_FilterIMsg(imsg))
        {
        switch (gtimsg->Class)
            {
            /* Act depending on the message */
            ...
            }
        /* Clean up the filtered message.  The return value is not needed */
        /* since we already have a pointer to the original message.      */
        GT_PostFilterIMsg(gtimsg);
        }
    }
/* other stuff can go here */
ReplyMsg(imsg);
```

You should not make any assumptions about the contents of the unfiltered **IntuiMessage** (**imsg** in the above example). Only two things are guaranteed: the unfiltered **IntuiMessage** must be replied to and the unfiltered **IntuiMessage** (if it produces anything when passed through **GT_FilterIMsg()**) will produce a meaningful GadTools **IntuiMessage** like those described in the section on the different kinds of gadgets. The relationship between the unfiltered and filtered messages are expected to change in the future. See the section on documented side-effects for more information.

DrawBevelBox()

A key visual signature shared by most GadTools gadgets is the raised or recessed bevelled box imagery. Since the program may wish to create its own boxes to match, GadTools provides the **DrawBevelBox()** and **DrawBevelBoxA()** functions.

```
void DrawBevelBoxA( struct RastPort *rport, long left, long top, long width, long height,
                    struct TagItem *taglist );
void DrawBevelBox ( struct RastPort *rport, long left, long top, long width, long height,
                    Tag tag1, ... );
```

The **rport** argument is a pointer to the **RastPort** into which the box is to be rendered. The **left**, **top**, **width** and **height** arguments specify the dimensions of the desired box.

The tag arguments, **tag1** or **taglist**, may be set as follows:

GT_VisualInfo (APTR)
The **VisualInfo** handle as returned by a prior call to **GetVisualInfo()**. This value is required.

GTBB_Recessed (BOOL)
A bevelled box may either appear to be raised to signify an area of the window that is selectable or recessed to signify an area of the window in which clicking will have no effect. Set this boolean tag to TRUE to get a recessed box. Omit this tag entirely to get a raised box.

DrawBevelBox() is a rendering operation, not a gadget. This means that the program must refresh any bevelled boxes rendered through this function if the window gets damaged.

GADGET KEYBOARD EQUIVALENTS

Often, users find it convenient to control gadgets using the keyboard. Starting with V37, it is possible to denote the keyboard equivalent for a GadTools gadget. The keyboard equivalent will be an underscored character in the gadget label, for easy identification. At the present time, however, the application is still responsible for implementing the reaction to each keypress.

Denoting a Gadget's Keyboard Equivalent

In order to denote the key equivalent, the application may add a marker-symbol to the gadget label. This is done by placing the marker-symbol immediately before the character to be underscored. This symbol can be any character that is not used in the label. The underscore character, '_' is the recommended marker-symbol. So, for example, to mark the letter "F" as the keyboard equivalent for a button labelled "Select Font...", create the gadget text:

```
ng.ng_GadgetText = "Select _Font...";
```

To inform GadTools of the underscore in the label, pass the GA_Underscore tag to **CreateGadget()** or **CreateGadgetA()**. The data-value associated with this tag is a character, not a string, which is the marker-symbol used in the gadget label:

```
GA_Underscore, '_',/* Note '_', not "_" !!! */
```

GadTools will create a gadget label which consists of the text supplied with the marker-symbol removed and the character following the marker-symbol underscored.

The gadget's label would look something like:

```
Select Font...
       -
```

Implementing a Gadget's Keyboard Equivalent Behavior

Currently, GadTools does not process keyboard equivalents for gadgets. It is up to the application writer to implement the correct behavior, normally by calling **GT_SetGadgetAttrs()** on the appropriate gadget. For some kinds of gadget, the behavior should be the same regardless of whether the keyboard equivalent was pressed with or without the shift key. For other gadgets, shifted and unshifted keystrokes will have different, usually opposite, effects.

Here is the correct behavior for keyboard equivalents for each kind of GadTools gadget:

Button Gadgets

The keyboard equivalent should invoke the same function that clicking on the gadget does. There is currently no way to highlight the button visuals programmatically when accessing the button through a keyboard equivalent.

Text-Entry and Number-Entry Gadgets

The keyboard equivalent should activate the gadget so the user may type into it. Use Intuition's **ActivateGadget()** call.

Checkbox Gadgets

The keyboard equivalent should toggle the state of the checkbox. Use **GT_SetGadgetAttrs()** and the GTCB_Checked tag.

Mutually-Exclusive Gadgets

The unshifted keystroke should activate the next choice, wrapping around from the last to the first. The shifted keystroke should activate the previous choice, wrapping around from the first to the last. Use **GT_SetGadgetAttrs()** and the GTMX_Active tag.

Cycle Gadgets

The unshifted keystroke should activate the next choice, wrapping around from the last to the first. The shifted keystroke should activate the previous choice, wrapping around from the first to the last. Use **GT_SetGadgetAttrs()** and the GTCY_Active tag.

Slider Gadgets

The unshifted keystroke should increase the slider's level by one, stopping at the maximum, while the shifted keystroke should decrease the level by one, stopping at the minimum. Use **GT_SetGadgetAttrs()** and the GTSL_Level tag.

Scroller Gadgets

The unshifted keystroke should increase the scroller's top by one, stopping at the maximum, while the shifted keystroke should decrease the scroller's top by one, stopping at the minimum. Use **GT_SetGadgetAttrs()** and the GTSC_Top tag.

Listview Gadgets

The unshifted keystroke should cause the next entry in the list to become the selected one, stopping at the last entry, while the shifted keystroke should cause the previous entry in the list to become the selected one, stopping at the first entry. Use **GT_SetGadgetAttrs()** and the GTLV_Top and GTLV_Selected tags.

Palette Gadgets

The unshifted keystroke should select the next color, wrapping around from the last to the first. The shifted keystroke should activate the previous color, wrapping around from the first to the last. Use **GT_SetGadgetAttrs()** and the GTPA_Color tag.

Text-Display and Number-Display Gadgets

These kinds of GadTools gadget have no keyboard equivalents since they are not selectable.

Generic Gadgets

Define appropriate keyboard functions based on the kinds of keyboard behavior defined for other GadTools kinds.

COMPLETE GADTOOLS GADGET EXAMPLE

Here's a working example showing how to set up and use a linked list of GadTools gadgets complete with keyboard shortcuts.

```
/* gadtoolsgadgets.c
** Simple example of using a number of gadtools gadgets.
**
** Compiled with SAS C v5.10a
** lc -b1 -cfistq -v -y gadtoolsgadgets
** blink FROM LIB:c.o gadtoolsgadgets.o TO gadtoolsgadgets LIB LIB:lc.lib LIB:amiga.lib
*/
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/gadgetclass.h>
#include <libraries/gadtools.h>

#include <clib/exec_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
#include <clib/gadtools_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)    { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* Gadget defines of our choosing, to be used as GadgetID's,
** also used as the index into the gadget array my_gads[].
*/
#define MYGAD_SLIDER     (0)
#define MYGAD_STRING1    (1)
#define MYGAD_STRING2    (2)
#define MYGAD_STRING3    (3)
#define MYGAD_BUTTON     (4)

/* Range for the slider: */
#define SLIDER_MIN  (1)
#define SLIDER_MAX (20)

struct TextAttr Topaz80 = { "topaz.font", 8, 0, 0, };

struct Library      *IntuitionBase;
struct Library      *GfxBase;
struct Library      *GadToolsBase;

/* Print any error message.  We could do more fancy handling (like
** an EasyRequest()), but this is only a demo.
*/
void errorMessage(STRPTR error)
{
if (error)
    printf("Error: %s\n", error);
}

/*
** Function to handle a GADGETUP or GADGETDOWN event.  For GadTools gadgets,
** it is possible to use this function to handle MOUSEMOVEs as well, with
** little or no work.
*/
VOID handleGadgetEvent(struct Window *win, struct Gadget *gad, UWORD code,
    WORD *slider_level, struct Gadget *my_gads[])
{
switch (gad->GadgetID)
    {
    case MYGAD_SLIDER:
        /* Sliders report their level in the IntuiMessage Code field: */
        printf("Slider at level %ld\n", code);
        *slider_level = code;
        break;
```

```
    case MYGAD_STRING1:
        /* String gadgets report GADGETUP's */
        printf("String gadget 1: '%s'.\n",
                ((struct StringInfo *)gad->SpecialInfo)->Buffer);
        break;
    case MYGAD_STRING2:
        /* String gadgets report GADGETUP's */
        printf("String gadget 2: '%s'.\n",
                ((struct StringInfo *)gad->SpecialInfo)->Buffer);
        break;
    case MYGAD_STRING3:
        /* String gadgets report GADGETUP's */
        printf("String gadget 3: '%s'.\n",
                ((struct StringInfo *)gad->SpecialInfo)->Buffer);
        break;
    case MYGAD_BUTTON:
        /* Buttons report GADGETUP's (button resets slider to 10) */
        printf("Button was pressed, slider reset to 10.\n");
        *slider_level = 10;
        GT_SetGadgetAttrs(my_gads[MYGAD_SLIDER], win, NULL,
                            GTSL_Level, *slider_level,
                            TAG_END);
        break;
    }
}


/*
** Function to handle vanilla keys.
*/
VOID handleVanillaKey(struct Window *win, UWORD code,
    WORD *slider_level, struct Gadget *my_gads[])
{
switch (code)
    {
    case 'v':
        /* increase slider level, but not past maximum */
        if (++*slider_level > SLIDER_MAX)
            *slider_level = SLIDER_MAX;
        GT_SetGadgetAttrs(my_gads[MYGAD_SLIDER], win, NULL,
                            GTSL_Level, *slider_level,
                            TAG_END);
        break;
    case 'V':
        /* decrease slider level, but not past minimum */
        if (--*slider_level < SLIDER_MIN)
            *slider_level = SLIDER_MIN;
        GT_SetGadgetAttrs(my_gads[MYGAD_SLIDER], win, NULL,
                            GTSL_Level, *slider_level,
                            TAG_END);
        break;
    case 'c':
    case 'C':
        /* button resets slider to 10 */
        *slider_level = 10;
        GT_SetGadgetAttrs(my_gads[MYGAD_SLIDER], win, NULL,
                            GTSL_Level, *slider_level,
                            TAG_END);
        break;
    case 'f':
    case 'F':
        ActivateGadget(my_gads[MYGAD_STRING1], win, NULL);
        break;
    case 's':
    case 'S':
        ActivateGadget(my_gads[MYGAD_STRING2], win, NULL);
        break;
    case 't':
    case 'T':
        ActivateGadget(my_gads[MYGAD_STRING3], win, NULL);
        break;
    }
}


/*
```

```
** Here is where all the initialization and creation of GadTools gadgets
** take place.  This function requires a pointer to a NULL-initialized
** gadget list pointer.  It returns a pointer to the last created gadget,
** which can be checked for success/failure.
*/
struct Gadget *createAllGadgets(struct Gadget **glistptr, void *vi,
    UWORD topborder, WORD slider_level, struct Gadget *my_gads[])
{
struct NewGadget ng;
struct Gadget *gad;

/* All the gadget creation calls accept a pointer to the previous gadget, and
** link the new gadget to that gadget's NextGadget field.  Also, they exit
** gracefully, returning NULL, if any previous gadget was NULL.  This limits
** the amount of checking for failure that is needed.  You only need to check
** before you tweak any gadget structure or use any of its fields, and finally
** once at the end, before you add the gadgets.
*/

/* The following operation is required of any program that uses GadTools.
** It gives the toolkit a place to stuff context data.
*/
gad = CreateContext(glistptr);

/* Since the NewGadget structure is unmodified by any of the CreateGadget()
** calls, we need only change those fields which are different.
*/
ng.ng_LeftEdge    = 140;
ng.ng_TopEdge     = 20+topborder;
ng.ng_Width       = 200;
ng.ng_Height      = 12;
ng.ng_GadgetText  = "_Volume:   ";
ng.ng_TextAttr    = &Topaz80;
ng.ng_VisualInfo  = vi;
ng.ng_GadgetID    = MYGAD_SLIDER;
ng.ng_Flags       = NG_HIGHLABEL;

my_gads[MYGAD_SLIDER] = gad = CreateGadget(SLIDER_KIND, gad, &ng,
                    GTSL_Min,         SLIDER_MIN,
                    GTSL_Max,         SLIDER_MAX,
                    GTSL_Level,       slider_level,
                    GTSL_LevelFormat, "%2ld",
                    GTSL_MaxLevelLen, 2,
                    GT_Underscore,    '_',
                    TAG_END);

ng.ng_TopEdge    += 20;
ng.ng_Height      = 14;
ng.ng_GadgetText  = "_First:";
ng.ng_GadgetID    = MYGAD_STRING1;
my_gads[MYGAD_STRING1] = gad = CreateGadget(STRING_KIND, gad, &ng,
                    GTST_String,    "Try pressing",
                    GTST_MaxChars, 50,
                    GT_Underscore, '_',
                    TAG_END);

ng.ng_TopEdge    += 20;
ng.ng_GadgetText  = "_Second:";
ng.ng_GadgetID    = MYGAD_STRING2;
my_gads[MYGAD_STRING2] = gad = CreateGadget(STRING_KIND, gad, &ng,
                    GTST_String,    "TAB or Shift-TAB",
                    GTST_MaxChars, 50,
                    GT_Underscore, '_',
                    TAG_END);

ng.ng_TopEdge    += 20;
ng.ng_GadgetText  = "_Third:";
ng.ng_GadgetID    = MYGAD_STRING3;
my_gads[MYGAD_STRING3] = gad = CreateGadget(STRING_KIND, gad, &ng,
                    GTST_String,    "To see what happens!",
                    GTST_MaxChars, 50,
                    GT_Underscore, '_',
                    TAG_END);

ng.ng_LeftEdge   += 50;
ng.ng_TopEdge    += 20;
```

```
ng.ng_Width      = 100;
ng.ng_Height     = 12;
ng.ng_GadgetText = "_Click Here";
ng.ng_GadgetID   = MYGAD_BUTTON;
ng.ng_Flags      = 0;
gad = CreateGadget(BUTTON_KIND, gad, &ng,
                    GT_Underscore, '_',
                    TAG_END);
return(gad);
}

/*
** Standard message handling loop with GadTools message handling functions
** used (GT_GetIMsg() and GT_ReplyIMsg()).
*/
VOID process_window_events(struct Window *mywin,
    WORD *slider_level, struct Gadget *my_gads[])
{
struct IntuiMessage *imsg;
ULONG imsgClass;
UWORD imsgCode;
struct Gadget *gad;
BOOL terminated = FALSE;

while (!terminated)
    {
    Wait (1 << mywin->UserPort->mp_SigBit);

    /* GT_GetIMsg() returns an IntuiMessage with more friendly information for
    ** complex gadget classes.  Use it wherever you get IntuiMessages where
    ** using GadTools gadgets.
    */
    while ((!terminated) &&
           (imsg = GT_GetIMsg(mywin->UserPort)))
        {
        /* Presuming a gadget, of course, but no harm...
        ** Only dereference this value (gad) where the Class specifies
        ** that it is a gadget event.
        */
        gad = (struct Gadget *)imsg->IAddress;

        imsgClass = imsg->Class;
        imsgCode = imsg->Code;

        /* Use the toolkit message-replying function here... */
        GT_ReplyIMsg(imsg);

        switch (imsgClass)
            {
            /*  --- WARNING --- WARNING --- WARNING --- WARNING --- WARNING ---
            ** GadTools puts the gadget address into IAddress of IDCMP_MOUSEMOVE
            ** messages.  This is NOT true for standard Intuition messages,
            ** but is an added feature of GadTools.
            */
            case IDCMP_GADGETDOWN:
            case IDCMP_MOUSEMOVE:
            case IDCMP_GADGETUP:
                handleGadgetEvent(mywin, gad, imsgCode, slider_level, my_gads);
                break;
            case IDCMP_VANILLAKEY:
                handleVanillaKey(mywin, imsgCode, slider_level, my_gads);
                break;
            case IDCMP_CLOSEWINDOW:
                terminated = TRUE;
                break;
            case IDCMP_REFRESHWINDOW:
                /* With GadTools, the application must use GT_BeginRefresh()
                ** where it would normally have used BeginRefresh()
                */
                GT_BeginRefresh(mywin);
                GT_EndRefresh(mywin, TRUE);
                break;
            }
        }
    }
}
```

```
/*
** Prepare for using GadTools, set up gadgets and open window.
** Clean up and when done or on error.
*/
VOID gadtoolsWindow(VOID)
{
struct TextFont *font;
struct Screen   *mysc;
struct Window   *mywin;
struct Gadget   *glist, *my_gads[4];
void            *vi;
WORD            slider_level = 5;
UWORD           topborder;

/* Open topaz 8 font, so we can be sure it's openable
** when we later set ng_TextAttr to &Topaz80:
*/
if (NULL == (font = OpenFont(&Topaz80)))
    errorMessage( "Failed to open Topaz 80");
else
    {
    if (NULL == (mysc = LockPubScreen(NULL)))
        errorMessage( "Couldn't lock default public screen");
    else
        {
        if (NULL == (vi = GetVisualInfo(mysc, TAG_END)))
            errorMessage( "GetVisualInfo() failed");
        else
            {
            /* Here is how we can figure out ahead of time how tall the  */
            /* window's title bar will be:                               */
            topborder = mysc->WBorTop + (mysc->Font->ta_YSize + 1);

            if (NULL == createAllGadgets(&glist, vi, topborder,
                                         slider_level, my_gads))
                errorMessage( "createAllGadgets() failed");
            else
                {
                if (NULL == (mywin = OpenWindowTags(NULL,
                        WA_Title,     "GadTools Gadget Demo",
                        WA_Gadgets,   glist,      WA_AutoAdjust,    TRUE,
                        WA_Width,       400,      WA_MinWidth,        50,
                        WA_InnerHeight, 140,      WA_MinHeight,       50,
                        WA_DragBar,    TRUE,      WA_DepthGadget,   TRUE,
                        WA_Activate,   TRUE,      WA_CloseGadget,   TRUE,
                        WA_SizeGadget, TRUE,      WA_SimpleRefresh, TRUE,
                        WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_REFRESHWINDOW |
                            IDCMP_VANILLAKEY | SLIDERIDCMP | STRINGIDCMP |
                            BUTTONIDCMP,
                        WA_PubScreen, mysc,
                        TAG_END)))
                    errorMessage( "OpenWindow() failed");
                else
                    {
                    /* After window is open, gadgets must be refreshed with a
                    ** call to the GadTools refresh window function.
                    */
                    GT_RefreshWindow(mywin, NULL);

                    process_window_events(mywin, &slider_level, my_gads);

                    CloseWindow(mywin);
                    }
                }
            /* FreeGadgets() even if createAllGadgets() fails, as some
            ** of the gadgets may have been created...If glist is NULL
            ** then FreeGadgets() will do nothing.
            */
            FreeGadgets(glist);
            FreeVisualInfo(vi);
            }
        UnlockPubScreen(NULL, mysc);
        }
    CloseFont(font);
    }
}
```

```
/*
** Open all libraries and run.  Clean up when finished or on error..
*/
void main(void)
{
if (NULL == (IntuitionBase = OpenLibrary("intuition.library", 37)))
    errorMessage( "Requires V37 intuition.library");
else
    {
    if (NULL == (GfxBase = OpenLibrary("graphics.library", 37)))
        errorMessage( "Requires V37 graphics.library");
    else
        {
        if (NULL == (GadToolsBase = OpenLibrary("gadtools.library", 37)))
            errorMessage( "Requires V37 gadtools.library");
        else
            {
            gadtoolsWindow();

            CloseLibrary(GadToolsBase);
            }
        CloseLibrary(GfxBase);
        }
    CloseLibrary(IntuitionBase);
    }
}
```

RESTRICTIONS ON GADTOOLS GADGETS

There is a strict set of functions and operations that are permitted on GadTools gadgets. Even if a technique is discovered that works for a particular case, be warned that it cannot be guaranteed and should not be used. If the trick concocted only works most of the time, it may introduce subtle problems in the future.

Never selectively or forcibly refresh gadgets. The only gadget refresh that should ever be performed is the initial **GT_RefreshWindow()** after a window is opened with GadTools gadgets attached. It is also possible to add gadgets after the window is opened by calling **AddGlist()** and **RefreshGlist()** followed by **GT_RefreshWindow()**. These refresh functions should not be called at any other time.

GadTools gadgets may not overlap with each other, with other gadgets or with other imagery. Doing this to modify the gadget's appearance is not supported.

GadTools gadgets may not be selectively added or removed from a window. This has to do with the number of Intuition gadgets that each call to **CreateGadget()** produces and with refresh constraints.

Never use **OnGadget()** or **OffGadget()** or directly modify the GFLG_DISABLED **Flags** bit. The only approved way to disable or enable a gadget is to use **GT_SetGadgetAttrs()** and the GA_Disabled tag. Those kinds of GadTools gadgets that do not support GA_Disabled may not be disabled (for now).

The application should never write into any of the fields of the **Gadget** structure or any of the structures that hang off it, with the exception noted earlier for GENERIC_KIND gadgets. Avoid making assumptions about the contents of these fields unless they are explicitly programmer fields (**GadgetID** and **UserData**, for example) or if they are guaranteed meaningful (**Left**, **Top**, **Width**, **Height**, **Flags**). On occasion, the program is specifically invited to read a field, for example the **StringInfo->Buffer** field.

GadTools gadgets may not be made relative sized or relative positioned. This means that the gadget flags GFLG_RELWIDTH, GFLG_RELHEIGHT, GFLG_RELBOTTOM and GFLG_RELRIGHT may not be specified. The activation type of the gadget may not be modified (for example changing GACT_IMMEDIATE to GACT_RELVERIFY). The imagery or the highlighting method may not be changed.

These restrictions are not imposed without reason; subtle or blatant problems may arise now or in future versions of GadTools for programs that violate these guidelines.

DOCUMENTED SIDE-EFFECTS

There are certain aspects of the behavior of GadTools gadgets that should not be depended on. This will help current applications remain compatible with future releases of the GadTools library.

When using **GT_FilterIMsg()** and **GT_PostFilterIMsg()**, never make assumptions about the message before or after filtering. I.e., do not interpret the unfiltered message, assume that it will or will not result in certain kinds of filtered message or assume it will result in a consumed message (i.e., when **GT_FilterIMsg()** returns NULL).

IDCMP_INTUITICKS messages are consumed when a scroller's arrows are repeating. That is, IDCMP_INTUITICKS will not be received while the user is pressing a scroller arrows. Do not depend or rely on this side effect, though, it will not necessarily remain so in the future.

A single call to **CreateGadget()** may create one or more actual gadgets. These gadgets, along with the corresponding code in GadTools, define the behavior of the particular kind of GadTools gadget requested. Only the behavior of these gadgets is documented, the number or type of actual gadgets is subject to change. Always refer to the gadget pointer received from **CreateGadget()** when calling **GT_SetGadgetAttrs()**. Never refer to other gadgets created by GadTools, nor create code which depends on their number or form.

For text-display gadgets, the **GTTX_CopyText** tag does not cause the text to be copied when the text is later changed with GTTX_Text.

The PLACETEXT **ng_Flags** are currently ignored by GENERIC_KIND gadgets. However, this may not always be so.

All GadTools gadgets set GADTOOL_TYPE in the gadget's **GadgetType** field. Do not use this flag to identify GadTools gadgets, as this flag is not guaranteed to be set in the future.

The palette gadget subdivides its total area into the individual color squares. Do not assume that the subdivision algorithm won't change.

Function Reference

The following are brief descriptions of the Intuition functions discussed in this chapter. See the "Amiga ROM Kernel Reference Manual: Includes and Autodocs" for details on each function call. All of these functions require Release 2 or a later version of the operating system.

Table 15-2: GadTools Library Functions

Function	Description
CreateGadgetA()	Allocate GadTools gadget, tag array form.
CreateGadget()	Allocate GadTools gadget, varargs form.
FreeGadgets()	Free all GadTools gadgets.
GT_SetGadgetAttrsA()	Update gadget, tag array form.
GT_SetGadgetAttrs()	Update gadget, varargs form.
CreateContext()	Create a base for adding GadTools gadgets.
CreateMenusA()	Allocate GadTools menu structures, tag array form.
CreateMenus()	Allocate GadTools menu structures, varargs form.
FreeMenus()	Free menus allocated with **CreateMenus()**.
LayoutMenuItemsA()	Format GadTools menu items, tag array form.
LayoutMenuItems()	Format GadTools menu items, varargs form.
LayoutMenusA()	Format GadTools menus, tag array form.
LayoutMenus()	Format GadTools menus, varargs form.
GT_GetIMsg()	GadTools gadget compatible version of **GetMsg()**.
GT_ReplyIMsg()	GadTools gadget compatible version of **ReplyMsg()**.
GT_FilterIMsg()	Process GadTools gadgets with **GetMsg()/ReplyMsg()**.
GT_PostFilterIMsg()	Process GadTools gadgets with **GetMsg()/ReplyMsg()**.
GT_RefreshWindow()	Display GadTools gadget imagery after creation.
GT_BeginRefresh()	GadTools gadget compatible version of **BeginRefresh()**.
GT_EndRefresh()	GadTools gadget compatible version of **EndRefresh()**.
DrawBevelBoxA()	Draw a 3D box, tag array form.
DrawBevelBox()	Draw a 3D box, varargs form.
GetVisualInfoA()	Get drawing information for GadTools, tag array form.
GetVisualInfo()	Get drawing information for GadTools, varargs form.
FreeVisualInfo()	Free drawing information for GadTools.

Chapter 16
ASL LIBRARY

This chapter describes the asl.library. The sole purpose of this library is to provide standard file and font requesters for application programs.

It is easier to understand the asl.library if you are familiar with some basic concepts of the Amiga operating system, especially **TagItem** arrays (described in the ''Utility Library'' chapter), Intuition screens and windows, graphics library font structures, and AmigaDOS pattern matching.

About Requesters

Requesters are temporary sub-windows used for confirming actions or selecting options. The most common type of requester is a file requester which is used to pick a file name for a load or save operation.

Under 1.3 (V34) and earlier versions of the Amiga operating system there was limited support for requesters. Intuition provides simple requesters which can be used to request responses such as OK or Cancel from the user. More elaborate Intuition requesters can be created by adding additional features such as string gadgets, however the result of this is that each application writer develops their own style of requester. Hence, the asl.library has been added to Release 2 of the Amiga operating system to make requesters more consistent. With asl.library, requesters are also much easier to create and take less memory.

> *The ASL Library Requires Release 2.* The asl.library requires Release 2 of the Amiga operating system, so only applications running under Release 2 and later versions of the Amiga OS can call its functions.

Requesters are very flexible and can be used for many different purposes. The Release 2 asl.library supports the two most common type of requesters:

- File requesters for choosing a file name in a load or save operation
- Font requesters for choosing a font in a text operation

Creating a File Requester

Opening an ASL requester requires the use of three functions:

```
APTR request =  AllocAslRequest( unsigned long type, struct TagItem *tagList );
BOOL success =  AslRequest( APTR request, struct TagItem *tagList );
void            FreeAslRequest( APTR request );
```

The first function you should call is **AllocAslRequest()**. This allocates the main data structure you will use, either a **FileRequester** structure or a **FontRequester** structure. You specify the type of requester you want for **AllocAslRequest()** by setting the type argument. This can be one of two values defined in *<libraries/asl.h>*: either ASL_FileRequest, to ask for a **FileRequester** structure, or ASL_FontRequest, to ask for a **FontRequester** structure.

Here's a listing of the **FileRequester** structure. (The **FontRequester** structure is discussed in more detail later in this chapter.)

```
struct FileRequester    {          /* (from <libraries/asl.h>)     */
        APTR    rf_Reserved1;
        BYTE    *rf_File;          /* Filename pointer             */
        BYTE    *rf_Dir;           /* Directory name pointer       */
        CPTR    rf_Reserved2;
        UBYTE   rf_Reserved3;
        UBYTE   rf_Reserved4;
        APTR    rf_Reserved5;
        WORD    rf_LeftEdge,rf_TopEdge; /* Preferred window pos   */
        WORD    rf_Width,rf_Height;     /* Preferred window size  */
        WORD    rf_Reserved6;
        LONG    rf_NumArgs;        /* A-la WB Args, for multiselects */
        struct WBArg *rf_ArgList;
        APTR    rf_UserData;       /* Applihandle (you may write!!)  */
        APTR    rf_Reserved7;
        APTR    rf_Reserved8;
        BYTE    *rf_Pat;           /* Pattern match pointer        */
        };                         /* note - more reserved fields follow */
```

Whichever requester type you use, you must allocate the requester structure with the **AllocAslRequest()** function call. Do not create the data structure yourself. The values in this structure are for *read access only*. Any changes to them must be performed only through asl.library function calls.

Once you have set up a requester structure with **AllocAslRequest()**, call **AslRequest()** to make the requester appear on screen. **AslRequest()** takes the requester data structure as an argument using it as a specification for the requester that it creates on screen.

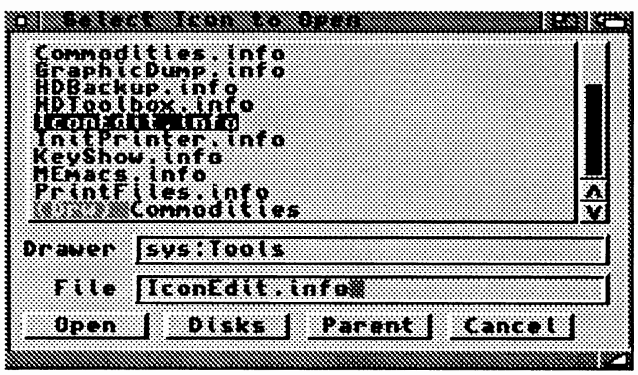


Figure 16-1: The ASL File Requester

AslRequest() is always synchronous to the calling program. That is, control does not return to your program until the user makes a selection or cancels. **AslRequest()** returns TRUE, if the user selects a file (or a font). In that case the file (or font) name that the user selected is returned in the requester data structure. **AslRequest()** returns FALSE if the user cancels the requester or the requester failed for some reason.

When you have finished with a requester use the **FreeAslRequest()** function to deallocate the requester data structure.

SPECIFYING REQUESTER OPTIONS WITH TAGITEMS

Both **AllocAslRequest()** and **AslRequest()** accept a **TagItem** array or tag list as an argument. The tag list is used to initialize or alter the values in the requester data structure.

A single **TagItem** consists of a tag name and an associated tag value. Tag items that apply to the asl.library are defined in *<libraries/asl.h>*. The basic tag items (used in the first example listed below) are:

Requester Tag Name	Used For
ASL_Hail	String to place in the title bar of the requester window
ASL_Width	Requester window width
ASL_Height	Requester window height
ASL_LeftEdge	Requester window y origin
ASL_TopEdge	Requester window x origin
ASL_OKText	String to place in OK gadget of requester
ASL_CancelText	String to place in Cancel gadget of requester
ASL_File	Default file name (for file requesters only)
ASL_Dir	Default directory name (for file requesters only)

Note that you are currently limited to about six characters for the replacement text if you use either the ASL_OKText or ASL_CancelText tags to change the text that appears in the OK and Cancel gadgets.

The contents of an ASL requester data structure are preserved across calls to **AslRequest()**. So, until the requester is freed, tag settings and user selections will remain in the data structure unless they are altered by tags in subsequent calls to **AslRequest()**. This is very useful because it allows the requester to remember and redisplay the user's previous selections. However, this also means that the programmer must assure that any addresses passed in ASL tags remain valid, or are refreshed on each call to **AslRequest()**.

Generally, options that you wish to specify only once, such as the initial position and size, should be specified as tags when you allocate the requester. Options that you wish to control for each use of the requester should be passed as tags each time the requester is opened with **AslRequest()**.

SIMPLE ASL FILE REQUESTER EXAMPLE

Here's a short example showing how to create a file requester with asl.library. If **AslRequest()** returns TRUE then the **rf_File** and **rf_Dir** fields of the requester data structure contain the name and directory of the file the user selected. Note that the user can type in the a name for the file and directory, which makes it possible for a file requester to return a file and directory that do not (currently) exist.

```
;/* filereq.c - Execute me to compile me with SASC 5.10
LC -b1 -cfistq -v -y -j73 filereq.c
Blink FROM LIB:c.o,filereq.o TO filereq LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
#include <exec/types.h>
#include <exec/libraries.h>
#include <libraries/asl.h>
#include <clib/exec_protos.h>
#include <clib/asl_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }     /* really */
#endif

UBYTE *vers = "$VER: filereq 37.0";

#define MYLEFTEDGE 0
#define MYTOPEDGE  0
#define MYWIDTH    320
#define MYHEIGHT   400

struct Library *AslBase = NULL;

struct TagItem frtags[] =
{
    ASL_Hail,       (ULONG)"The RKM file requester",
    ASL_Height,     MYHEIGHT,
    ASL_Width,      MYWIDTH,
    ASL_LeftEdge,   MYLEFTEDGE,
    ASL_TopEdge,    MYTOPEDGE,
    ASL_OKText,     (ULONG)"O KAY",
    ASL_CancelText, (ULONG)"not OK",
    ASL_File,       (ULONG)"asl.library",
    ASL_Dir,        (ULONG)"libs:",
    TAG_DONE
};

void main(int argc, char **argv)
{
    struct FileRequester *fr;

    if (AslBase = OpenLibrary("asl.library", 37L))
    {
        if (fr = (struct FileRequester *)
            AllocAslRequest(ASL_FileRequest, frtags))
        {
            if (AslRequest(fr, NULL))
            {
                printf("PATH=%s  FILE=%s\n", fr->rf_Dir, fr->rf_File);
                printf("To combine the path and filename, copy the path\n");
                printf("to a buffer, add the filename with Dos AddPart().\n");
            }
            FreeAslRequest(fr);
        }
        else printf("User Cancelled\n");

        CloseLibrary(AslBase);
    }
}
```

FILE PATTERN MATCHING AND MULTIPLE SELECTS

A file requester can filter out certain file and directory entries using the ''wildcard'' feature of AmigaDOS. To activate the wildcard feature for a file requester, you use the ASL_FuncFlags tag. Each bit in the ASL_FuncFlags tag item controls a special option of the requester depending on its type (file or font). See *<libraries/asl.h>* for a complete listing of the options that the ASL_FuncFlags tag controls.

File Requester Flag	Used For
FILF_PATGAD	Enables the file name pattern matching gadget
FILF_MULTISELECT	Enables multiple selection of files
FILF_NEWIDCMP	Use separate IDCMP for requester sharing a custom screen (see below)
FILF_SAVE	Makes the file requester a *save* requester (see below)

If the FILF_PATGAD bit of the ASL_FuncFlags tag is set, the file requester will appear with a "Pattern" gadget in addition to the usual file name and directory name gadgets. The user can type an AmigaDOS wildcard pattern into this gadget and the pattern will be used to limit the file names that appear in the requester. An application can also supply a default pattern using the ASL_Pattern tag item. A hidden unchangeable pattern can be created by supplying an ASL_Pattern without a FILF_PATGAD gadget. Such invisible patterns should not be used if there is any reason that the user may need to access a file which does not match the pattern.

Another feature of the ASL file requester is multiple selection. When multiple selection is enabled, the user can choose more than one file name in a single directory by selecting names in the requester's scrolling list gadget with the mouse. This option, like pattern matching, is set up with the ASL_FuncFlags tag.

If the FILF_MULTISELECT bit of the ASL_FuncFlags tag is set, the file requester will allow multiple selection. When the user selects several file names through the multiple selection feature, the FileRequester's **rf_NumArgs** field contains the number of files selected and the **rf_ArgList** field contains a pointer to an array of **WBArg** structures (defined in *<workbench/startup.h>*). There is a **WBArg** structure containing a file name for each file the user selected.

The following example illustrates a file requester with both a pattern matching gadget and multiple selection enabled.

```
;/* filepat.c - Execute me to compile me with SASC 5.10
LC -b1 -cfistq -v -y -j73 filepat.c
Blink FROM LIB:c.o,filepat.o TO filepat LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>
#include <graphics/displayinfo.h>
#include <libraries/asl.h>
#include <workbench/startup.h>

#include <clib/asl_protos.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }     /* really */
#endif

UBYTE *vers = "$VER: filepat 37.0";

struct Library *AslBase = NULL;
struct Library *IntuitionBase = NULL;
struct Screen *screen = NULL;
struct Window *window = NULL;

void main(int argc, char **argv)
{
    struct FileRequester *fr;
    struct WBArg *frargs;
    int x;
```

```
    if (AslBase = OpenLibrary("asl.library", 37L))
    {
        if (IntuitionBase = (struct IntuitionBase *)
                OpenLibrary("intuition.library", 37L))
        {
            if (screen = (struct Screen *)OpenScreenTags(NULL,
                    SA_DisplayID, HIRESLACE_KEY,
                    SA_Title, "ASL Test Screen",
                    TAG_END))
            {
                if (window = (struct Window *)OpenWindowTags(NULL,
                        WA_CustomScreen, screen,
                        WA_Title, "Demo Customscreen, File Pattern, Multi-select",
                        WA_Flags, WINDOWDEPTH | WINDOWDRAG,
                        TAG_END))
                {
                    if (fr = (struct FileRequester *)
                        AllocAslRequestTags(ASL_FileRequest,
                            ASL_Hail, (ULONG)"FilePat/MultiSelect Demo",
                            ASL_Dir,  (ULONG)"libs:",
                            ASL_File, (ULONG)"asl.library",

                            /* Initial pattern string for pattern matching */
                            ASL_Pattern, (ULONG)"~(rexx#?|math#?)",

                            /* Enable multiselection and pattern match gadget */
                            ASL_FuncFlags, FILF_MULTISELECT | FILF_PATGAD,

                            /* This requester comes up on the screen of this
                            ** window (and uses window's message port, if any).
                            */
                            ASL_Window, window,
                            TAG_DONE))
                    {
                        /* Put up file requester */
                        if (AslRequest(fr, 0L))
                        {
                            /* If the file requester's rf_NumArgs field
                            ** is not zero, the user multiselected. The
                            ** number of files is stored in rf_NumArgs.
                            */
                            if (fr->rf_NumArgs)
                            {
                                /* rf_ArgList is an array of WBArg structures
                                ** (see <workbench/startup.h>). Each entry in
                                ** this array corresponds to one of the files
                                ** the user selected (in alphabetical order).
                                */
                                frargs = fr->rf_ArgList;

                                /* The user multiselected, step through
                                ** the list of selected files.
                                */
                                for ( x=0;  x < fr->rf_NumArgs;  x++ )
                                    printf("Argument %d: PATH=%s FILE=%s\n",
                                        x, fr->rf_Dir, frargs[x].wa_Name);
                            }
                            else
                                /* The user didn't multiselect, use the
                                ** normal way to get the file name.
                                */
                                printf("PATH=%s FILE=%s\n", fr->rf_Dir, fr->rf_File);
                        }
                        /* Done with the FileRequester, better return it */
                        FreeAslRequest(fr);
                    }
                    CloseWindow(window);
                }
                CloseScreen(screen);
            }
            CloseLibrary(IntuitionBase);
        }
        CloseLibrary(AslBase);
    }
}
```

The previous example demonstrates two alternate functions for creating and using ASL requesters:

```
APTR AllocAslRequestTags( unsigned long type, Tag Tag1, ... );
BOOL AslRequestTags( APTR request, Tag Tag1, ... );
```

AllocAslRequestTags() can be used instead of **AllocAslRequest()** to allocate and set up the file requester. This is an *amiga.lib* function that will accept **TagItems** directly in its parameter list, rather than a pointer to an array of **TagItems**.

Similarly, **AslRequestTags()** will accept **TagItems** directly instead of requiring a pointer to an array of **TagItems** as **AslRequest()** does.

ASL REQUESTERS AND CUSTOM SCREENS

An application that uses a custom screen normally wants its requesters to open on its screen. Using the ASL_Window tag, a program can associate a requester with a specific window so that the requester appears on the same screen as the window. The ASL_Window tag is followed by a pointer to a window structure. ASL_Window works with both file and font requesters. The example above shows how the ASL_Window tag is used with a file requester.

Normally, a requester associated with a window (using ASL_Window) shares that window's IDCMP port for its communication. An application may not want to share an IDCMP port with the requester. Using the ASL_FuncFlags tag, a program can ask for a requester that creates its own IDCMP port. There are two flags that accomplish this. The first, FILF_NEWIDCMP, is used on file requesters. The other, FONF_NEWIDCMP, is used on font requesters.

THE SAVE REQUESTER

The save requester is a special type of file requester used for save operations. It differs from the regular ASL file requester in several ways. First, the color of the text making up the file names and the background color are interchanged. This makes it more apparent to the user that they are looking at a save requester (instead of the usual load requester).

Another difference, is that a save requester does not allow the user to select an existing file name by double-clicking on an entry in the scrolling list gadget. This helps prevent the user from accidentally overwriting the wrong file.

Save requesters can also create directories. If the user types a directory name into the save requester and the directory doesn't exist, the save requester will create that directory (after getting the user's permission via another requester).

To create a save requester, set the FILF_SAVE flag of the ASL_FuncFlags tag. Remember that ASL tags and flag values are preserved across calls to **AslRequest()**, so if you use a save requester, you must clear the FILF_SAVE bit and reset your ASL_FuncFlags when you want a load requester. Note that it does not make sense to have multiselection in a save requester, so the FILF_SAVE flag overrides the FILF_MULTISELECT flag.

THE DIRECTORY REQUESTER

Sometimes a program may only require a directory name from the user. There is another variation on asl.library's file requester that allows this. The ASL_ExtFlags1 tag contains a flag bit to toggle this option. If the FIL1F_NOFILES flag of ASL_ExtFlags1 is set, the requester will appear without a string gadget for file names and will display only directory names in the scrolling list gadget. When **AslRequest()** (or **AslRequestTags()**) returns successfully, the **rf_Dir** field of the **FileRequester** structure contains the name of the directory the user selected.

Another flag defined for ASL_ExtFlags1 is FIL1F_MATCHDIRS. If file pattern matching is on (see the FILF_PATGAD flag for ASL_FuncFlags), setting FIL1F_MATCHDIRS tells the file requester to pattern match directory names as well as file names. Of course, if both of these ASL_ExtFlags1 flags are set, the requester will only pattern match directory names.

Creating a Font Requester

The ASL library also contains a font requester. Using the font requester is very similar to using the file requester. First, allocate a requester structure with **AllocAslRequest()** or **AllocAslRequestTags()**. The type should be set to **ASL_FontRequest** in order to get a **FontRequester** structure:

```
struct FontRequester    {
        APTR     fo_Reserved1[2];
        struct TextAttr fo_Attr;         /* Returned TextAttr           */
        UBYTE    fo_FrontPen;            /* Returned pens, if selected  */
        UBYTE    fo_BackPen;
        UBYTE    fo_DrawMode;
        APTR     fo_UserData;
        /* missing from asl.h but present in this structure */
        SHORT    fo_LeftEdge, fo_TopEdge, fo_Width, fo_Height;
        };
```

Once the requester is set up, call **AslRequest()** or **AslRequestTags()** to make the requester appear on screen. These functions return TRUE if the user makes a selection. In that case, the font selected is returned as a **TextAttr** structure in the **fo_Attr** field of the **FontRequester** structure. (The **TextAttr** structure is defined in *<graphics/text.h>*. See the *Amiga ROM Kernel Manual: Includes and Autodocs* for a complete listing.) If the user cancels the font requester FALSE is returned.

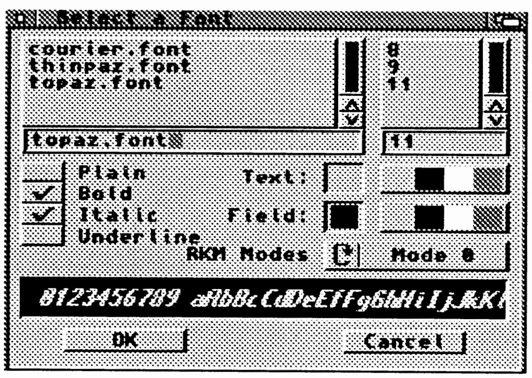


Figure 16-2: The ASL Font Requester

When the requester is no longer needed, call **FreeAslRequest()** to deallocate the requester data structure.

SPECIFYING FONT REQUESTER OPTIONS WITH TAGITEMS

As with a file requester, the font requester is specified with a **TagItem** list. There are several tags that are specific to the font requester:

Font Requester Tag Name	Used For
ASL_FontName	Default font (**fo_Attr.ta_Name**)
ASL_FontHeight	Default font size (**fo_Attr.ta_YSize**)
ASL_FontStyles	Default font style (**fo_Attr.ta_Style**)
ASL_FontFlags	Default font flags (**fo_Attr.ta_Flags**)
ASL_FrontPen	Default font color (**fo_FrontPen**)
ASL_BackPen	Default font background color (**fo_BackPen**)
ASL_ModeList	Alternate strings for the drawing mode gadget (see below)
ASL_MinHeight	Minimum font height the requester will display
ASL_MaxHeight	Maximum font height the requester will display

Note that the last two tags only limit the range of font sizes that the font requester displays, the user is free to type in any value.

Font requesters have additional special options that are controlled through the ASL_FuncFlags tag. This tag works the same way as it does with file requesters but with different options available. Recall that the data for this tag is divided into bit fields, each of which controls a requester option. The flags used with the ASL_FuncFlags tag in a font requester are defined in *<libraries/asl.h>*:

Font Requester Flags	Used For
FONF_FRONTCOLOR	Enables font color selection gadgets
FONF_BACKCOLOR	Enables font background color selection gadget
FONF_STYLES	Enables font style selection gadget
FONF_FIXEDWIDTH	Limits display to fixed width fonts only
FONF_DRAWMODE	Enables font draw mode gadget

A simple font requester (one without any of the above FONF_ flags set) only lets the user choose a font and a Y size. Setting the flags above adds options to the font requester. FONF_FRONTCOLOR and FONF_BACKCOLOR add color selection gadgets to the requester, one for choosing a font's foreground color (labeled "Text") and the other for choosing the background color (labeled "Field"). The font requester records the user's setting in the **FontRequester**'s **fo_FrontPen** and **fo_BackPen** fields.

FONF_STYLES sets up several gadgets to choose the style of the font (bold, italics, underline). The font requester saves these settings in the **fo_Attr.ta_Style** bit field according to the style flags defined in *<graphics/text.h>*. FONF_FIXEDWIDTH limits the font name display to fixed width (non-proportional) fonts (note that this does not prevent the user from typing in a proportional font name).

FONF_DRAWMODE adds a cycle gadget to the font requester so the user can choose the draw mode. The draw mode is saved in the requester's **fo_DrawMode** field. The number stored there corresponds to the draw mode's position in the gadget's cycle.

The draw mode cycle gadget initially is labeled "Mode" and has three elements in its cycle: "JAM1", "JAM2", and "Complement". These yield a result of 0, 1, and 2, respectively. It is possible to change the names and number of draw modes with the ASL_ModeList tag. This tag accepts a pointer to an array of strings. The first string replaces "Mode" as the label for the draw mode cycle gadget. The strings that follow replace the elements of the cycle gadget. The last entry in the array has to be NULL to tell the requester where the list of entries ends.

EXAMPLE FONT REQUESTER

The following example illustrates how to use a font requester.

```
;/* fontreq.c - Execute me to compile me with Lattice 5.10
LC -b1 -cfistq -v -y -j73 fontreq.c
Blink FROM LIB:c.o,fontreq.o TO fontreq LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <libraries/asl.h>

#include <clib/asl_protos.h>
#include <clib/exec_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }     /* really */
#endif

UBYTE *vers = "$VER: fontreq 37.0";

struct Library *AslBase = NULL;

/* Our replacement strings for the "mode" cycle gadget.  The
** first string is the cycle gadget's label.  The other strings
** are the actual strings that will appear on the cycle gadget.
*/
UBYTE *modelist[] =
{
    "RKM Modes",
    "Mode 0",
    "Mode 1",
    "Mode 2",
    "Mode 3",
    "Mode 4",
    NULL
};


void main(int argc, char **argv)
{
    struct FontRequester *fr;

    if (AslBase = OpenLibrary("asl.library", 37L))
    {
        if (fr = (struct FontRequester *)
            AllocAslRequestTags(ASL_FontRequest,
                /* tell the requester to use my custom mode names */
                ASL_ModeList, modelist,

                /* Supply initial values for requester */
                ASL_FontName, (ULONG)"topaz.font",
                ASL_FontHeight, 11L,
                ASL_FontStyles, FSF_BOLD | FSF_ITALIC,
                ASL_FrontPen,  0x00L,
                ASL_BackPen,   0x01L,

                /* Only display font sizes between 8 and 14, inclusive. */
                ASL_MinHeight, 8L,
                ASL_MaxHeight, 14L,

                /* Give all the gadgetry, but only display fixed width fonts */
                ASL_FuncFlags, FONF_FRONTCOLOR | FONF_BACKCOLOR |
                    FONF_DRAWMODE | FONF_STYLES | FONF_FIXEDWIDTH,
                TAG_DONE))
        {
            /* Pop up the requester */
            if (AslRequest(fr, NULL))
            {
                /* The user selected something,  report their choice */
```

```
            printf("%s\n  YSize = %d  Style = 0x%x   Flags = 0x%x\n"
                   "  FPen = 0x%x   BPen = 0x%x   DrawMode = 0x%x\n",
                           fr->fo_Attr.ta_Name,
                           fr->fo_Attr.ta_YSize,
                           fr->fo_Attr.ta_Style,
                           fr->fo_Attr.ta_Flags,
                           fr->fo_FrontPen,
                           fr->fo_BackPen,
                           fr->fo_DrawMode);
        }
        else
            /* The user cancelled the requester, or some kind of error
            ** occurred preventing the requester from opening. */
            printf("Request Cancelled\n");
        FreeAslRequest(fr);
    }
    CloseLibrary(AslBase);
  }
}
```

Calling Custom Functions from a Requester

The ASL_HookFunc tag passes an ASL requester a pointer to a custom function. The requester can use this function for two purposes. The first is to determine if the requester should display a particular file or font name. The other purpose is to process messages that the requester receives at its IDCMP port that are not meant for the requester. Hook functions are set up through flag values used with the ASL_FuncFlags tag:

Hook Function Flag	Used For
FILF_DOWILDFUNC	Call user hook function on each name in a file requester
FONF_DOWILDFUNC	Call user hook function on each name in a font requester
FILF_DOMSGFUNC	Call user hook function for IDCMP messages not used by a file requester
FONF_DOMSGFUNC	Call user hook function for IDCMP messages not used by a font requester

The FILF_DOWILDFUNC and FONF_DOWILDFUNC flags cause a requester to call the function you specify with the ASL_HookFunc tag for every file or font entry. The requester displays the file or font name only if your hook function tells it to. For a file requester, if your hook function returns a zero, the file requester will display the file name. For a font requester, if your hook function returns anything but zero, the font requester will display the font name and size.

The FILF_DOMSGFUNC and FONF_DOMSGFUNC flags cause a requester to call your hook function whenever it receives an IntuiMessage that it cannot use at the IDCMP port that it shares with your window. (See the section on "ASL Requesters and Custom Screens" earlier in this chapter for more information about sharing IDCMP ports.) If the requester receives any messages that are not meant for the requester it will call your hook function (specified with the ASL_HookFunc tag). Your hook function is responsible for returning a pointer to the IntuiMessage. The requester will take care of replying to the message.

PARAMETERS PASSED TO CUSTOM HOOK FUNCTIONS

A requester always passes three parameters to your custom hook function:

```
ULONG MyHookFunc(ULONG type, CPTR object, CPTR AslRequester)
```

If **MyHookFunc()** is called from a file requester doing _DOWILDFUNC, the three parameters are:

> `type` = FILF_DOWILDFUNC
> `object` = pointer to an **AnchorPath** structure (from *<dos/dosasl.h>*)
> `AslRequester` = pointer to the **FileRequester** that called the hook function
> (Return a zero to display this file)

The **AnchorPath** structure is a dos.library structure used in pattern matching. Refer to the *AmigaDOS Manual, 3rd Edition* by Bantam Books for more information.

If **MyHookFunc()** is called from a font requester doing _DOWILDFUNC, the three parameters are:

> `type` = FONF_DOWILDFUNC
> `object` = pointer to a **TextAttr** structure (from *<graphics/text.h>*)
> `AslRequester` = pointer to the **FontRequester** that called the hook function
> (Return non-zero to display this particular font size)

If **MyHookFunc()** is called from a file or font requester doing _DOMSGFUNC, the three parameters are:

> `type` = FILF_DOMSGFUNC (file requester) or FONF_DOMSGFUNC (font requester)
> `object` = pointer to the **IntuiMessage** for the function to process
> `AslRequester` = pointer to the **FileRequester** or **FontRequester** that called the hook function
> (Return a pointer to the **IntuiMessage**)

Notice that it is possible for a requester to use both _DOWILDFUNC and _DOMSGFUNC at the same time. Your hook function has to differentiate between the two cases by testing the type passed to it. It is not possible for a font and file requester to share a hook function for a _DOWILDFUNC, because FILF_DOWILDFUNC is defined to be the same value as FONF_DOWILDFUNC, so the hook function cannot tell if the object (from the prototype above) is a pointer to an **AnchorPath** structure or a pointer to a **TextAttr** structure. It is possible for font and file requesters to share one hook function for _DOMSGFUNC (even though FILF_DOMSGFUNC and FONF_DOMSGFUNC are equal) because, in this case, font and file requesters both call your hook function in the same manner.

EXAMPLE ASL REQUESTER WITH CUSTOM HOOK FUNCTION

The following example illustrates the use of a hook function for both _DOWILDFUNC and _DOMSGFUNC.

```
;/* filehook.c - Execute me to compile me with Lattice 5.10
LC -b1 -cfistq -v -y -j73 filehook.c
Blink FROM LIB:c.o,filehook.o TO filehook LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <dos/dosasl.h>
#include <libraries/asl.h>
```

```
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/asl_protos.h>
#include <clib/intuition_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)      { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }     /* really */
#endif

#define DESTPATLENGTH 20

UBYTE *vers = "$VER: filehook 37.0";

CPTR HookFunc();

struct Library *AslBase = NULL;
struct Library *IntuitionBase = NULL;
struct Window  *window = NULL;

/* this is the pattern matching string that the hook function uses */
UBYTE *sourcepattern = "(#?.info)";
UBYTE pat[DESTPATLENGTH];

void main(int argc, char **argv)
{
    struct FileRequester *fr;

    if (AslBase = OpenLibrary("asl.library", 37L))
    {
        if (IntuitionBase = (struct IntuitionBase *)
                    OpenLibrary("intuition.library", 37L))
        {
            /* This is a V37 dos.library function that turns a pattern matching
            ** string into something the DOS pattern matching functions can
            ** understand.
            */
            ParsePattern(sourcepattern, pat, DESTPATLENGTH);

            /* open a window that gets ACTIVEWINDOW events */
            if (window = (struct Window *)OpenWindowTags(NULL,
                    WA_Title, "ASL Hook Function Example",
                    WA_IDCMP, IDCMP_ACTIVEWINDOW,
                    WA_Flags, WFLG_DEPTHGADGET,
                    TAG_END))
            {
                if (fr = AllocFileRequest())
                {
                    if (AslRequestTags(fr,
                        ASL_Dir, (ULONG)"SYS:Utilities",
                        ASL_Window, window,
                        ASL_TopEdge, 0L,
                        ASL_Height, 200L,
                        ASL_Hail, (ULONG)"Pick an icon, select save",
                        ASL_HookFunc, (ULONG)HookFunc,
                        ASL_FuncFlags, FILF_DOWILDFUNC | FILF_DOMSGFUNC | FILF_SAVE,
                        ASL_OKText, (ULONG)"Save",
                        TAG_DONE))
                    {
                        printf("PATH=%s FILE=%s\n", fr->rf_Dir, fr->rf_File);
                        printf("To combine the path and filename, copy the path\n");
                        printf("to a buffer, add the filename with Dos AddPart().\n");
                    }
                    FreeFileRequest(fr);
                }
                CloseWindow(window);
            }
            CloseLibrary(IntuitionBase);
        }
        CloseLibrary(AslBase);
    }
}
```

```
CPTR HookFunc(LONG type, CPTR obj, struct FileRequester *fr)
{
    static BOOL returnvalue;
    switch(type)
    {
        case FILF_DOMSGFUNC:
        /* We got a message meant for the window */
            printf("You activated the window\n");
            return(obj);
            break;
        case FILF_DOWILDFUNC:
        /* We got an AnchorPath structure, should
        ** the requester display this file? */

            /* MatchPattern() is a dos.library function that
            ** compares a matching pattern (parsed by the
            ** ParsePattern() DOS function) to a string and
            ** returns true if they match. */
            returnvalue = MatchPattern(pat,
                    ((struct AnchorPath *)obj)->ap_Info.fib_FileName);

            /* we have to negate MatchPattern()'s return value
            ** because the file requester expects a zero for
            ** a match not a TRUE value */
            return( (CPTR)(! returnvalue) );
            break;
    }
}
```

Function Reference

The following are brief descriptions of the ASL library functions. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call. All of these functions require Release 2 or a later version of the operating system.

Table 16-1: Functions for ASL Requesters

Function	Description
AllocAslRequest()	Allocates an ASL font or file requester from a **TagItem** array
AllocAslRequestTags()	Same as **AllocAslRequest()** but accepts tags directly
AslRequest()	Displays an ASL requester with options set up in a **TagItem** array
AslRequestTags()	Same as **AslRequest()** but accepts tags directly
FreeAslRequest()	Deallocates an ASL requester created with **AllocAslRequest()**

Chapter 17
INTRODUCTION TO EXEC

The *Multitasking Executive*, better known as *Exec*, is the heart of the Amiga's operating system. All other systems in the Amiga rely on it to control multitasking, to manage the message-based interprocess communications system, and to arbitrate access to system resources. Because just about every software entity on the Amiga (including application programs) needs to use Exec in some way, every Amiga programmer has to have a basic understanding of its fundamentals.

Multitasking

A conventional micro-computer spends a lot of its time waiting for things to happen. It has to wait for such things as the user to push buttons on the keyboard or mouse, for data to come in through the serial port, and for data to go out to a disk drive. To make efficient use of the CPU's time, an operating system can have the CPU carry out some other task while it is waiting for such events to occur.

A *multitasking* operating system reduces the amount of time it wastes, by switching to another program when the current one needs to wait for an event. A multitasking operating system can have several programs, or tasks, running at the same time. Each task runs independently of the others, without having to worry about what the other tasks are doing. From a task's point of view, it's as if each task has a computer all to itself.

The Amiga's multitasking works by switching which task is currently using the CPU. A task can be a user's application program, or it can be a task that controls system resources (like the disk drives or the keyboard). Each task has a priority assigned to it. Exec will let the task with the highest priority use the CPU, but only if the task is ready to run. A task can be in one of three states: *ready*, *sleeping*, or *running*.

A ready task is not currently using the CPU but is waiting to use the processor. Exec keeps a list of the tasks that are ready. Exec sorts this list according to task priority, so Exec can easily find the ready task with the highest priority. When Exec switches the task that currently has control of the CPU, it switches to the task at the top of this list.

A sleeping task is not currently running and is waiting for some event to happen. When that event occurs, Exec will move the sleeping task into the list of ready tasks.

A running task is currently using the CPU. It will remain the current task until one of three things occur:

- A higher priority task becomes ready, so the OS preempts the current task and switches to the higher priority task.
- The currently running task needs to wait for an event, so it goes to sleep and Exec switches to the highest priority task in Exec's ready list.
- The currently running task has had control of the CPU for at least a preset time period called a *quantum* and there is another task of equal priority ready to run. In this case, Exec will preempt the current task for the ready one with the same priority. This is known as *time-slicing*. When there is a group of tasks of equal priority on the top of the ready list, Exec will cycle through them, letting each one use the CPU for a quantum (a slice of time).

The terms "task" and "process" are often used interchangeably to represent the generic concept of task. On the Amiga, this terminology can be a little confusing because of the names of the data structures that are associated with Exec tasks. Each task has a structure associated with it called a **Task** structure (defined in *<exec/tasks.h>*). Most application tasks use a superset of the **Task** structure called a **Process** structure (defined in *<dos/dosextens.h>*). These terms are confusing to Amiga programmers because there is an important distinction between the Exec task with only a **Task** structure and an Exec task with a **Process** structure.

The **Process** structure builds on the **Task** structure and contains some extra fields which allow the DOS library to associate an AmigaDOS environment to the task. Some elements of a DOS environment include a current input and output stream and a current working directory. These elements are important to applications that need to do standard input and output using functions like **printf()**.

Exec only pays attention to the **Task** structure portion of the **Process** structure, so, as far as Exec is concerned, there is no difference between a task with a **Task** structure and a task with a **Process** structure. Exec considers both of them to be tasks.

An application doesn't normally worry about which structure their task uses. Instead, the system that launches the application takes care of it. Both Workbench and the Shell (CLI) attach a **Process** structure to the application tasks that they launch.

Dynamic Memory Allocation

The Amiga has a soft machine architecture, meaning that all tasks, including those that are part of its operating system, do not use fixed memory addresses. As a result, any program that needs to use a chunk of memory must allocate that memory from the operating system.

There are two functions on the Amiga for simple memory allocation: **AllocMem()** and **AllocVec()**. The two functions accept the same parameters, a ULONG containing the size of the memory block in bytes followed by 32-bit specifier for memory attributes. Both functions return the address of a longword aligned memory block if they were successful or NULL if something went wrong.

AllocVec() differs from **AllocMem()** in that it records the size of the memory block allocated so an application does not have to remember the size of a memory block it allocated. **AllocVec()** was introduced in Release 2, so it is not available to the 1.3 developer.

Normally the bitmask of memory attributes passed to these functions will contain any of the following attributes (these flags are defined in <*exec/memory.h*>):

MEMF_ANY
This indicates that there is no requirement for either Fast or Chip memory. In this case, while there is Fast memory available, Exec will only allocate Fast memory. Exec will allocate Chip memory if there is not enough Fast memory.

MEMF_CHIP
This indicates the application wants a block of Chip memory, meaning it wants memory addressable by the Amiga custom chips. Chip memory is required for any data that will be accessed by custom chip DMA. This includes floppy disk buffers, screen memory, images that will be blitted, sprite data, copper lists, and audio data. If your application requires a block of Chip RAM, it must use this flag to allocate the Chip RAM. Otherwise, the application will fail on machines with expanded memory.

MEMF_FAST
This indicates a memory block outside of the range that the Amiga's custom chips can access. The "FAST" in MEMF_FAST has to do with the custom chips and the CPU trying to access the same memory at the same time. Because the custom chips and the CPU both have access to Chip RAM, the CPU may have to wait to access Chip RAM while some custom chip is reading or writing Chip RAM. In the case of Fast RAM, the custom chips do not have access to it, so the CPU does not have to contend with the custom chips access to Fast RAM, making CPU accesses to Fast RAM generally faster than CPU access to Chip RAM.

Since the flag specifies memory that the custom chips cannot access, this flag is mutually exclusive with the MEMF_CHIP flag. If you specify the MEMF_FAST flag, your allocation will fail on Amigas that have only Chip memory. Use MEMF_ANY if you would *prefer* Fast memory.

MEMF_PUBLIC
This indicates that the memory should be accessible to other tasks. Although this flag doesn't do anything right now, using this flag will help ensure compatibility with possible future features of the OS (like virtual memory and memory protection).

MEMF_CLEAR
This indicates that the memory should be initialized with zeros.

If an application does not specify any attributes when allocating memory, the system first looks for MEMF_FAST, then MEMF_CHIP. There are additional memory allocation flags for Release 2: MEM_LOCAL, MEMF_24BITDMA and MEMF_REVERSE. See the Exec Autodoc for **AllocMem()** in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* or the include file <*exec/memory.h*> for additional information on these flags. Use of these flags under earlier versions of the operating system will cause your allocation to fail.

> *Make Sure You Have Memory.* Always check the result of any memory allocation to be sure the type and amount of memory requested is available. Failure to do so will lead to trying to use an non-valid pointer.

When an application is finished with a block of memory it allocated, it must return it to the operating system. There is a function to return memory for both the **AllocMem()** and the **AllocVec()** functions. **FreeMem()** releases memory allocated by **AllocMem()**.

It takes two parameters, a pointer to a memory block and the size of the memory block. **FreeVec()** releases memory allocated by **AllocVec()**. It takes only one parameter, a pointer to a memory block allocated by **AllocVec()**. The following example shows how to allocate and deallocate memory.

```
APTR  my_mem;

if (my_mem = AllocMem(100, MEMF_ANY))
{
    /* Your code goes here */
    FreeMem(my_mem, 100);
}
else  { /* couldn't get memory, exit with an error */ }
```

Signals

The Amiga uses a mechanism called *signals* to tell a task that some event occurred. Each task has its own set of 32 signals, 16 of which are set aside for system use. When one task signals a second task, it asks the OS to set a specific bit in the 32-bit long word set aside for the second task's signals.

Signals are what makes it possible for a task to go to sleep. When a task goes to sleep, it asks the OS to wake it up when a specific signal bit gets set. That bit is tied to some event. When that event occurs, that signal bit gets set. This triggers the OS into waking up the sleeping task.

To go to sleep, a task calls a system function called **Wait()**. This function takes one argument, a bitmask that tells Exec which of the task's signal bits to "listen to". The task will only wake up if it receives one of the signals whose corresponding bit is set in that bitmask. For example, if a task wanted to wait for signals 17 and 19, it would call **Wait()** like this:

```
mysignals = Wait(1L<<17 | 1L<<19);
```

Wait() puts the task to sleep and will not return until some other task sets at least one of these two signals. When the task wakes up, **mysignals** will contain the bitmask of the signal or signals that woke up the task. It is possible for several signals to occur simultaneously, so any combination of the signals that the task **Wait()**ed on can occur. It is up to the waking task to use the return value from **Wait()** to figure out which signal or signals occurred.

Looking for Break Keys

One common usage of signals on the Amiga is for processing a user break. As was mentioned earlier, the OS reserves 16 of a tasks 32 signals for system use. Four of those 16 signals are used to tell a task about the Control-C, D, E, and F break keys. An application can process these signals. Usually, only CLI-based programs receive these signals because the Amiga's console handler is about the only user input source that sets these signals when it sees the Control-C, D, E, and F key presses.

The signal masks for each of these key presses are defined in *<dos/dos.h>*:

```
SIGBREAKF_CTRL_C
SIGBREAKF_CTRL_D
SIGBREAKF_CTRL_E
SIGBREAKF_CTRL_F
```

Note that these are bit masks and not bit numbers.

Processing Signals without Wait()ing

In some cases an application may need to process signals but cannot go to sleep to wait for them. For example, a compiler might want to check to see if the user hit Control-C, but it can't to go to sleep to check for the break because that will stop the compiler. In this case, the task can periodically check its own signal bits for the Ctrl-C break signal using the Exec library function, **SetSignal()**:

```
oldsignals = ULONG SetSignal(ULONG newsignals, ULONG signalmask);
```

Although **SetSignal()** can change a task's signal bits, it can also monitor them. The following fragment illustrates using **SetSignal()** to poll a task's signal bits for a Ctrl-C break:

```
/* Get current state of signals */
signals = SetSignal(0L, 0L);

/* check for Ctrl-C */
if (signals & SIGBREAKF_CTRL_C)
{
    /* The Ctrl-C signal has been set, take care of processing it... */

    /* ...then clear the Ctrl-C signal */
    SetSignal(0L, SIGBREAKF_CTRL_C);
}
```

If your task is waiting for signals, but is also waiting for other events that have no signal bit (such as input characters from standard input), you may need to use **SetSignal()**. In such cases, you must be careful not to poll in a tight loop (also known as busy-waiting). Busy-waiting hogs CPU time and degrades the performance of other tasks. One easy way around this is for a task to sleep briefly within its polling loop by using the timer.device, or the graphics function **WaitTOF()**, or (if the task is a **Process**) the DOS library **Delay()**) or **WaitForChar()** functions.

For more information on signals, see the "Exec Signals" chapter of this manual.

Interprocess Communications

Another feature of the Amiga OS is its system of message-based *interprocess communication*. Using this system, a task can send a message to a message port owned by another task. Tasks use this mechanism to do things like trigger events or share data with other tasks, including system tasks. Exec's message system is built on top of Exec's task signaling mechanism. Most Amiga applications programming (especially Intuition programming) relies heavily upon this message-based form of interprocess communication.

When one task sends a message to another task's message port, the OS adds the message to the port's message queue. The message stays in this queue until the task that owns the port is ready to check its port for messages. Typically, a task has put itself to sleep while it is waiting for an event, like a message to arrive at its message port. When the message arrives, the task wakes up to look in its message port. The messages in the message port's queue are arranged in first-in-first-out (FIFO) order so that, when a task receives several messages, it will see the messages in the order they arrived at the port.

A task can use a message to share any kind of data with another task. This is possible because a task does not actually transmit an entire message, it only passes a pointer to a message. When a task creates a message (which can have many Kilobytes of data attached to it) and sends it to another task, the actual message does not move.

Essentially, when task A sends a message to task B, task A lends task B a chunk of its memory, the memory occupied by the message. After task A sends the message, it has relinquished that memory to task B, so it cannot touch the memory occupied by the message. Task B has control of that memory until task B returns the message to task A with the **ReplyMsg()** function.

Let's look at an example. Many applications use Intuition windows as a source for user input. Without getting into too much detail about Intuition, when an application opens a window, it can set up the window so Intuition will send messages about certain user input events. Intuition sends these messages to a message port created by Intuition for use with this window. When an application successfully opens a window, it receives a pointer to a **Window** structure, which contains a pointer to this message port (**Window.UserPort**). For this example, we'll assume the window has been set up so Intuition will send a message only if the user clicks the window's close gadget.

When Intuition opens the window in this example, it creates a message port for the task that opened the **Window**. Because the most common message passing system uses signals, creating this message port involves using one of the example task's 32 signals. The OS uses this signal to signal the task when it receives a message at this message port. This allows the task to sleep while waiting for a "Close Window" event to arrive. Since this simple example is only waiting for activity at one message port, it can use the **WaitPort()** function. **WaitPort()** accepts a pointer to a message port and puts a task to sleep until a message arrives at that port.

This simple example needs two variables, one to hold the address of the window and the other to hold the address of a message.

```
struct Message *mymsg; /*defined in <exec/ports.h> */
struct Window *mywin;  /* defined in <intuition/intuition.h> */
        ...

/* at some point, this application would have successfully opened a */
/* window and stored a pointer to it in mywin.                      */
        ...

        /* Here the application goes to sleep until the user clicks the window's close    */
        /* gadget.  This window was set up so that the only time Intuition will send a    */
        /* message to this window's port is if the user clicks the window's close gadget. */

        WaitPort(mywin->UserPort);
        while (mymsg = GetMsg(mywin->UserPort))
            /* process message now or copy information from message */
            ReplyMsg(mymsg);
        ...

/* Close window, clean up */
```

The Exec function **GetMsg()** is used to extract messages from the message port. Since the memory for these messages belongs to Intuition, the example relinquishes each message as it finishes with them by calling **ReplyMsg()**. Notice that the example keeps on trying to get messages from the message port until **mymsg** is NULL. This is to make sure there are no messages left in the message port's message queue. It is possible for several messages to pile up in the message queue while the task is asleep, so the example has to make sure it replies to all of them. Note that the window should never be closed within this **GetMsg()** loop because the **while** statement is still accessing the window's **UserPort**.

Note that each task with a **Process** structure (sometimes referred to as a process) has a special process message port, **Process.pr_MsgPort**. This message port is only for use by Workbench and the DOS library itself. *No application should use this port for its own use!*

Waiting on Message Ports and Signals at the Same Time

Most applications need to wait for a variety of messages and signals from a variety of sources. For example, an application might be waiting for **Window** events and also timer.device messages. In this case, an application must **Wait()** on the combined signal bits of all signals it is interested in, including the signals for the message ports where any messages might arrive.

The **MsgPort** structure, which is defined in *<exec/ports.h>*, is what Exec uses to keep track of a message port. The **UserPort** field from the example above points to one of these structures. In this structure is a field called **mp_SigBit**, which contains the *number* (not the actual bit mask) of the message port's signal bit. To **Wait()** on the signal of a message port, **Wait()** on a bit mask created by shifting 1L to the left **mp_SigBit** times (1L << **msgport->mp_SigBit**). The resulting bit mask can be OR'd with the bit masks for any other signals you wish to simultaneously wait on.

Libraries and Devices

One of the design goals for the Amiga OS was to make the system dynamic enough so that the OS could be extended and updated without effecting existing applications. Another design goal was to make it easy for different applications to be able to share common pieces of code. The Amiga accomplished these goals through a system of libraries. An Amiga library consists of a collection of related functions which can be anywhere in system memory (RAM or ROM).

Devices are very similar to libraries, except they usually control some sort of I/O device and contain some extra standard functions. Although this section does not really discuss devices directly, the material here applies to them. For more information on devices, see the "Exec Device I/O" section of this manual or the *Amiga ROM Kernel Reference Manual: Devices.*

An application accesses a library's functions through the library's jump, or vector, table. Before a task can use the functions of a particular library, it must first acquire the library's base pointer by calling the Exec **OpenLibrary()** function:

```
struct Library *OpenLibrary( UBYTE *libName, unsigned long mylibversion );
```

where **libName** is a string naming the library and **mylibversion** is a version number for the library. The version number reflects a revision of the system software. The chart below lists the specific Amiga OS release versions that system libraries versions correspond to:

```
30      Kickstart 1.0  - This revision is obsolete.
31      Kickstart 1.1  - This was an NTSC only release and is obsolete.
32      Kickstart 1.1  - This was a PAL only release and is obsolete.
33      Kickstart 1.2  - This is the oldest revision of the OS still in use.

34      Kickstart 1.3  - This is almost the same as release 1.2 except it has
                         an Autobooting expansion.library

35      This is a special RAM-loaded version of the 1.3 revision, except
                         that it knows about the A2024 display modes.  No
                         application should need this library unless they
                         need to open an A2024 display mode under 1.3.

36      Kickstart 2.0  - This is the original Release 2 revision that was
                         initially shipped on early Amiga 3000 models.

37      Kickstart 2.04 - This is the general Release 2 revision for all
                         Amiga models.
```

The **OpenLibrary()** function looks for a library with a name that matches **libName** and also with a version at least as high as **mylibversion**. For example, to open version 33 or greater of the Intuition library:

```
IntuitionBase = OpenLibrary("intuition.library", 33L);
```

In this example, if version 33 or greater of the Intuition library is not available, **OpenLibrary()** returns NULL. A version of zero in **OpenLibrary()** tells the OS to open any version of the library. Unless your code requires Release 2 features, it should specify a version number of 33 to remain backwards compatible with Kickstart 1.2.

When **OpenLibrary()** looks for a library, it first looks in memory. If the library is not in memory, **OpenLibrary()** will look for the library on disk. If **libName** is a library name with an absolute path (for example, "myapp:mylibs/mylib.library"), **OpenLibrary()** will follow that absolute path looking for the library. If **libName** is only a library name ("diskfont.library"), **OpenLibrary()** will look for the library in the directory that the LIBS: logical assign currently references.

If **OpenLibrary()** finds the library on disk, it takes care of loading it and initializing it. As part of the initialization process, **OpenLibrary()** dynamically creates a jump, or vector, table. There is a vector for each function in the library. Each entry in the table consists of a 680x0 jump instruction (JMP) to one of the library functions. The OS needs to create the vector table dynamically because the library functions can be anywhere in memory.

After the library is loaded into memory and initialized, **OpenLibrary()** can actually "open" the library. It does this by calling the library's **Open** function vector. Every library has a standard vector set aside for an OPEN function so the library can set up any data or processes that it needs. Normally, a library's OPEN function increments its open count to keep track of how many tasks currently have the library opened.

If any step of **OpenLibrary()** fails, it returns a NULL value. If **OpenLibrary()** is successful, it returns the address of the library base. The library base is the address of this library's **Library** structure (defined in *<exec/libraries.h>*). The **Library** structure immediately follows the vector table in memory.

After an application is finished with a library, it must close it by calling **CloseLibrary()**:

```
void CloseLibrary(struct Library *libPtr);
```

where **libPtr** is a pointer to the library base returned when the library was opened with **OpenLibrary()**.

LIBRARY VECTOR OFFSETS (LVOs)

After an application has successfully opened a library, it can start using the library's functions. To access a library function, an application needs the library base address returned by **OpenLibrary()** and the function's *Library Vector Offset (LVO)*. A function's LVO is the offset from the library's base address to the function's vector in the vector table, which means an LVO is a negative number (the vectors precedes the library base in memory). Application code enters a library function by doing a jump to subroutine (the 680x0 instruction JSR) to the proper negative offset (LVO) from the address of the library base. The library vector itself is a jump instruction (the 680x0 instruction JMP) to the actual library function which is somewhere in memory (see Figure 17-1: An Exec Library Base in RAM).

The only legal way for an application to access a library function is through the vector table. A function's LVO is always the same on every system and is not subject to change. A function's jump vector can, and does, change. Assuming that a function's jump vector is static is *very bad*, so don't do it.

Each library has four vectors set aside for library housekeeping: OPEN, CLOSE, EXPUNGE, and RESERVED. The OPEN vector points to a function that performs any custom initialization that this library needs, for example, opening other libraries that this library uses. The CLOSE function takes care of any clean up necessary when an application closes a library. The EXPUNGE vector points to a function that prepares the library for removal from the system. Normally the OS will not remove a library from memory until the system needs the memory the library occupies. The other vector, RESERVED, does not do anything at present and is reserved for future system use.

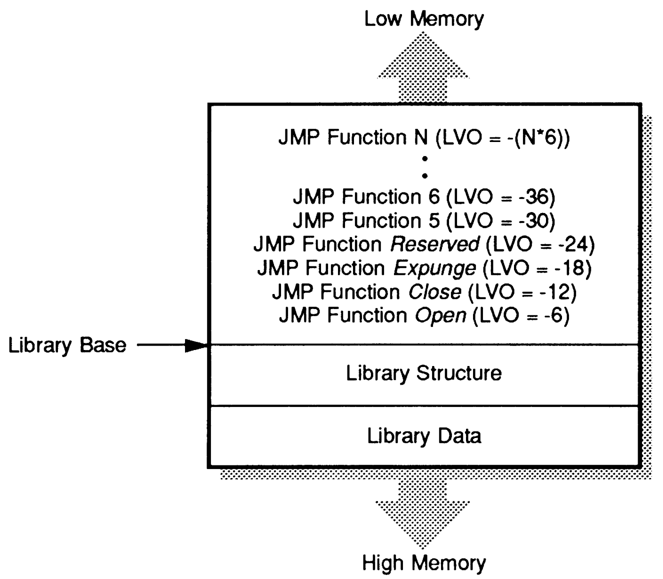


Figure 17-1: An Exec Library Base in RAM

CALLING A LIBRARY FUNCTION

To call a function in an Amiga system library, an assembler application must put the library's base address in register A6 and JSR to the function's LVO relative to A6. For example to call the Intuition function **DisplayBeep()**:

```
;************************************************************************
        xref    _LVODisplayBeep
;...
        move.l  A6, -(sp)              ;save the current contents of A6
;---- intuition.library must already be opened
;---- and IntuitionBase must contain its base address
        move.l  IntuitionBase, A6      ;put intuition base pointer in A6
        jsr     _LVODisplayBeep(A6)    ;call DisplayBeep()
        move.l  (SP)+, A6              ;restore A6 to its original value
```

IntuitionBase contains a pointer to the Intuition library's library base and **_LVODisplayBeep** is the LVO for the **DisplayBeep()** function. The external reference (xref) to **_LVODisplayBeep** is resolved from the linker library, *amiga.lib*. This linker library contains the LVO's for all of the standard Amiga libraries. Each LVO label starts with "_LVO" followed by the name of the library function.

> *System Functions Do Not Preserve D0, D1, A0 and A1.* If you need to preserve D0, D1, A0, or A1 between calls to system functions, you will have to save and restore these values yourself. Amiga system functions use these registers as scratch registers and may write over the values your program left in these registers. The system functions preserve the values of all other registers. The result of a system function, if any, is returned in D0.

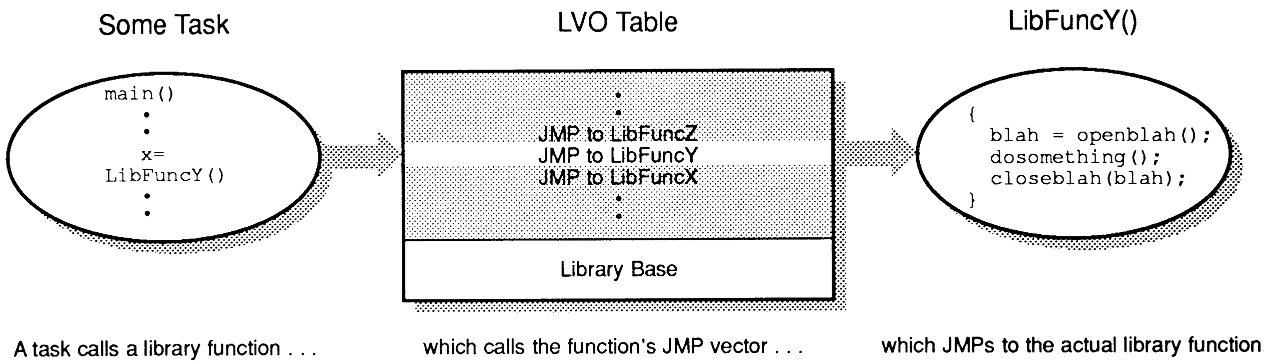


Figure 17-2: Calling a Library Function

The example above is the actual assembly code generated by the macro named **LINKLIB**, which is defined in *<exec/libraries.i>*. The following fragment performs the same function as the fragment above:

```
LINKLIB _LVODisplayBeep, IntuitionBase
```

The *amiga.lib* linker library also contains small functions called *stubs* for each function in the Amiga OS. These stubs are normally for use with C code.

Function parameters in C are normally pushed on the stack when a program calls a function. This presents a bit of a problem for the C programmer when calling Amiga OS functions because the Amiga OS functions expect their parameters to be in specific CPU registers. Stubs solve this problem by copying the parameters from the stack to the appropriate register.

For example, the Autodoc for the Intuition library function **MoveWindow()** shows which registers **MoveWindow()** expects its parameters to be:

```
MoveWindow(window, deltaX, deltaY);
           A0      D0      D1
```

The stub for **MoveWindow()** in *amiga.lib* has to copy **window** to register A0, **deltaX** to register D0, and **deltaY** to register D1.

The stub also copies Intuition library base into A6 and does an address-relative JSR to **MoveWindow()**'s LVO (relative to A6). The stub gets the library base from a global variable in your code called **IntuitionBase.** If you are using the stubs in *amiga.lib* to call Intuition library functions, you must declare a global variable called **IntuitionBase**. It must be called **IntuitionBase** because *amiga.lib* is specifically looking for the label **IntuitionBase**.

```
/* This global declaration is here so amiga.lib can find
   the intuition.library base pointer.
*/
struct Library *IntuitionBase;

        ...

void main(void)
{
        ...

        /* initialize IntuitionBase */
```

```
        if (IntuitionBase = OpenLibrary("intuition.library", 33L))
        {
                ...

                /* When this code gets linked with amiga.lib, the linker
                   extracts the DisplayBeep() stub routine from amiga.lib
                   and copies it into the executable.  The stub copies whatever
                   is in the variable IntuitionBase into A6, and JSRs to
                   _LVODisplayBeep(A6).
                */
                DisplayBeep();

                ...

                CloseLibrary(IntuitionBase);
        }
        ...
}
```

There is a specific label in *amiga.lib* for the library base of every library in the Amiga operating system. The chart below lists the names of the library base pointer *amiga.lib* associates with each Amiga OS library. The labels for library bases are also listed in the "Function Offsets Reference" list in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Library Name	Library Base Pointer Name
asl.library	AslBase
commodities.library	CxBase
diskfont.library	DiskfontBase
†dos.library	DOSBase
†exec.library	SysBase
expansion.library	ExpansionBase
gadtools.library	GadToolsBase
graphics.library	GfxBase
icon.library	IconBase
iffparse.library	IFFParseBase
intuition.library	IntuitionBase
keymap.library	KeyMapBase
layers.library	LayersBase
mathffp.library	MathBase
mathieeedoubbas.library	MathIeeeDoubBasBase
mathieeedoubtrans.library	MathIeeeDoubTransBase
mathieeesingbas.library	MathIeeeSingBasBase
mathieeesingtrans.library	MathIeeeSingTransBase
mathtrans.library	MathTransBase
rexxsys.library	RexxSysBase
rexxsupport.library	RexxSupBase
translator.library	TranslatorBase
utility.library	UtilityBase
version.library	(system private)
workbench.library	WorkbenchBase
†Automatically opened by the standard C startup module	

Table 17-1: Amiga.lib Library Base Labels

The chart mentions that **SysBase** and **DOSBase** are already set up by the standard C startup module. For more information on the startup module, see the "Workbench and Icon Library" chapter of this manual.

You May Not Need amiga.lib. Many C compilers provide ways of using *pragmas* or *registerized parameters*, so that a C program does not have to link with an *amiga.lib* stub to access a library function. See your compiler documentation for more details.

Chapter 18
EXEC LIBRARIES

Exec maintains lists of *libraries* and *devices*. An Amiga library consists of a collection of related functions which can be anywhere in system memory (RAM or ROM). An Amiga device is very similar to an Amiga library, except that a device normally controls some sort of I/O hardware, and generally contains a limited set of standard functions which receive commands for controlling I/O. For more information on how to use devices for I/O, see the "Exec Device I/O" chapter of this book.

> *Not for Beginners.* This chapter concentrates on the internal workings of Exec libraries (and devices). Most application programmers will not to know the internals workings of libraries to program the Amiga. For an introduction to libraries and how to use them, see chapter one, "Introduction to Amiga System Libraries".

What is a Library?

A library consists of a group of functions somewhere in memory (ROM or RAM), a vector table, and a **Library** structure which can be followed by an optional private data area for the library. The library's base pointer (as returned by **OpenLibrary()**) points to the library's **Library** data structure:

```
struct Library
{
    struct  Node lib_Node;
    UBYTE   lib_Flags;
    UBYTE   lib_pad;
    UWORD   lib_NegSize;          /* number of bytes before library */
    UWORD   lib_PosSize;          /* number of bytes after library */
    UWORD   lib_Version;
    UWORD   lib_Revision;
    APTR    lib_IdString;
    ULONG   lib_Sum;              /* the checksum itself */
    UWORD   lib_OpenCnt;          /* number of current opens */
};
/* Meaning of the flag bits: */
                              /* A task is currently running a checksum on */
#define LIBF_SUMMING (1 << 0)  /* this library (system maintains this flag) */
#define LIBF_CHANGED (1 << 1)  /* One or more entries have been changed in the library code */
                              /* vectors used by SumLibrary (system maintains this flag)  */

#define LIBF_SUMUSED (1 << 2)  /* A checksum fault should cause a system panic (library flag) */
#define LIBF_DELEXP (1 << 3)   /* A user has requested expunge but another user still has */
                              /* the library open (this is maintained by library)       */
```

USING A LIBRARY TO REFERENCE DATA

Most libraries (such as Intuition, graphics and Exec) have other data that follows the **Library** data structure in memory. Although it is not normally necessary, a program can use the library base pointer to access the **Library** structure and any custom library data.

In general, the system's library base data is read-only, and should be directly accessed as little as possible, primarily because the format of the data may change in future revisions of the library. If the library provides functions to allow access to library data, use those instead.

RELATIONSHIP OF LIBRARIES TO DEVICES

A device is a software specification for hardware control based on the **Library** structure. The structures of libraries and devices are so similar that the routine **MakeLibrary()** is used to construct both.

Devices require the same initial four code vectors as a library, but must have two additional code vectors for beginning and terminating special device I/O commands. The I/O commands that devices are expected to perform, at minimum, are shown in the "Exec Device I/O" chapter. An example device is listed in the *Amiga ROM Kernel Reference Manual: Devices.*

MINIMUM SUBSET OF LIBRARY VECTORS

The first four code vectors of a library must be the following entries:

OPEN
: is the entry point called by the function **OpenLibrary()**. In most libraries, OPEN increments the library variable **lib_OpenCnt**. This variable is also used by CLOSE and EXPUNGE.

CLOSE
: is the entry point called by the function **CloseLibrary()**. It decrements the library variable **lib_OpenCnt** and may do a delayed EXPUNGE.

EXPUNGE
: prepares the library for removal from the system. This often includes deallocating memory resources that were reserved during initialization. EXPUNGE not only frees the memory allocated for data structures, but also the areas reserved for the library node itself.

RESERVED
: is a fourth function vector reserved for future use. It must always return zero.

CHANGING THE CONTENTS OF A LIBRARY

The way in which an Amiga library is organized allows a programmer to change where the system looks for a library routine. Exec provides a function to do this: **SetFunction()**. The **SetFunction()** routine redirects a library function call to an application-supplied function. (Although it's not addressed here, **SetFunction()** can also be used on Exec devices.) For instance, the AmigaDOS command SetPatch uses **SetFunction()** to replace some OS routines with improved ones, primarily to fix bugs in ROM libraries.

The format of the **SetFunction()** routine is as follows:

```
SetFunction( struct Library *lib, LONG funcOffset, APTR funcEntry)
                            A1            A0                D0
```

The **lib** argument is a pointer to the library containing the function entry to be changed. The **funcOffset** is the Library Vector Offset (negative) of the function and **funcEntry** is the address of the new function you want to replace it with. The **SetFunction()** routine replaces the entry in the library's vector table at the given Library Vector Offset with a new address that points to the new routine and returns the old vector address. The old address can be used in the new routine to call the original library function.

Normally, programs should not attempt to ''improve'' library functions. Because most programmers do not know exactly what system library functions do internally, OS patches can do more harm than good. However, a legitimate use for **SetFunction()** is in a debugger utility. Using **SetFunction()**, a debugger could reroute system calls to a debugging routine. The debugging routine can inspect the arguments to a library function call before calling the original library function (if everything is OK). Such a debugger doesn't do any OS patching, it merely inspects.

> *SetFunction() is for Advanced Users Only.* It is very difficult to cleanly exit after performing **SetFunction()** because other tasks may be executing your code and also because additional **SetFunction()**'s may have occurred on the same function. Also note that certain libraries (for example the V33 version of DOS library) and some individual library function vectors are of non-standard format and cannot be replaced via **SetFunction()**.

Although useful, performing **SetFunction()** on a library routines poses several problems. If a second task performs **SetFunction()** on the same library entry, **SetFunction()** returns the address of the new routine to the second task, *not* the original system vector. In that case, the first task can no longer exit cleanly since that would leave the second task with an invalid pointer to a function which it could be relying on.

You also need to know when it is safe to unload your replacement function. Removing it while another task is executing it will quickly lead to a crashed system. Also, the replacement function will have to be re-entrant, like all Exec library functions.

> *Don't Do This!* For those of you who might be thinking about writing down the ROM addresses returned by **SetFunction()** and using them in some other programs: *Forget It.* The address returned by **SetFunction()** is only good on the current system at the current time.

Adding a Library

Exec provides several ways to add your own libraries to the system library list. One rarely used way is to call **LoadSeg()** (a DOS library function) to load your library and then use the Exec **MakeLibrary()** and **AddLibrary()** functions to initialize your library and add it to the system.

MakeLibrary() allocates space for the code vectors and data area, initializes the library node, and initializes the data area according to your specifications, returning to you a library base pointer. The base pointer may then be passed to **AddLibrary()** to add your library to the system.

Another way to initialize and add a library or device to the system is through the use of a **Resident** structure or *romtag* (see <exec/resident.h>). A romtag allows you to place your library or device in a directory (default LIBS: for libraries, DEVS: for devices) and have the OS automatically load and initialize it when an application tries to open it with **OpenLibrary()** or **OpenDevice()**.

Two additional initialization methods exist for a library or device which is bound to a particular Amiga expansion board. The library or device (containing a romtag) may be placed in the SYS:Expansion drawer, along with an icon containing the Manufacturer and Product number of the board it requires. If the startup-sequence BindDrivers command finds that board in the system, it will load and initialize the matching Expansion drawer device or library. In addition, since 1.3, the Amiga system software supports ROM drivers on expansion boards. See the "Expansion Library" chapter for additional information on ROM drivers and Expansion drawer drivers. The sample device code in the *Amiga ROM Kernel Reference Manual: Devices* volume of this manual set may be conditionally assembled as an Expansion drawer driver.

RESIDENT (ROMTAG) STRUCTURE

A library or device with a romtag should start with `MOVEQ #-1,D0` (to safely return an error if a user tries to execute the file), followed by a **Resident** structure:

```
STRUCTURE RT,0
   UWORD RT_MATCHWORD        * romtag identifier (==$4AFC)
   APTR  RT_MATCHTAG         * pointer to the above UWORD (RT_MATCHWORD)
   APTR  RT_ENDSKIP          * usually ptr to end of your code
   UBYTE RT_FLAGS            * usually RTF_AUTOINIT
   UBYTE RT_VERSION          * release version number (for example: 37)
   UBYTE RT_TYPE             * type of module (NT_LIBRARY)
   BYTE  RT_PRI              * initialization priority (for example: 0)
   APTR  RT_NAME             * pointer to node name ("my.library")
   APTR  RT_IDSTRING         * pointer to id string ("name ver.rev (date)")
   APTR  RT_INIT             * pointer to init code or AUTOINIT tables
   LABEL RT_SIZE             * size of a Resident structure (romtag)
```

If you wish to perform **MakeLibrary()** and **AddLibrary()** yourself, then your RT_FLAGS will not include RTF_AUTOINIT, and RT_INIT will be simply be a pointer to your own initialization code. To have Exec automatically load and initialize the library, set the RTF_AUTOINIT flag in the **Resident** structure's RT_FLAGS field, and point RT_INIT to a set four longwords containing the following:

dataSize
: This is the size of your library data area, i.e., the combined size of the standard **Library** node structure plus your own library-specific data.

vectors
: This is a pointer to a table of pointers to your library's functions, terminated with a -1. If the first word of the table is -1, then the table is interpreted as a table of words specifying the relative displacement of each function entry point from the start of the table. Otherwise it is treated as a table of longword address pointers to the functions. **vectors** must specify a valid table address.

structure
: This parameter points to the base of an **InitStruct()** data region. That is, it points to the first location within a table that the **InitStruct()** routine can use to initialize your **Library** node structure, library-specific data, and other memory areas. **InitStruct()** will typically be used to initialize the data segment of the library, perhaps forming data tables, task control blocks, I/O control blocks, etc. If this entry is a 0, then **InitStruct()** is not called.

initFunction
: This points to a routine that is to be executed after the library (or device) node has been allocated and the code and data areas have been initialized. When the routine is called, the base address of the newly created library is passed in D0. If **initFunction** is zero, no initialization routine is called.

Complete source code for an RT_AUTOINIT library may be found in the appendix of this book.

Chapter 19
EXEC DEVICE I/O

The Amiga system devices are software engines that provide access to the Amiga hardware. Through these devices, a programmer can operate a modem, spin a disk drive motor, time an event, and blast a trumpet sound in stereo. Yet, for all that variety, the programmer uses each device in the same manner.

What is a Device?

An Amiga device is a software module that accepts commands and data and performs I/O operations based on the commands it receives. In most cases, it interacts with either internal or external hardware, (the exceptions are the clipboard device and ramdrive device which simply use memory). Generally, an Amiga device runs as a separate task which is capable of processing your commands while your application attends to other things.

Table 19-1: Amiga System Devices

Amiga Device	Purpose
Audio	Controls the use of the audio hardware.
Clipboard	Manages the cutting and pasting of common data blocks
Console	Provides the line-oriented user interface.
Gameport	Controls the two mouse/joystick ports.
Input	Processes input from the gameport and keyboard devices.
Keyboard	Controls the keyboard.
Narrator	Produces the Amiga synthesized speech.
Parallel	Controls the parallel port.
Printer	Converts a standard set of printer control codes to printer specific codes.
SCSI	Controls the Small Computer Standard Interface hardware.
Serial	Controls the serial port.
Timer	Provides timing functions to measure time intervals and send interrupts.
Trackdisk	Controls the Amiga floppy disk drives.

The philosophy behind the devices is that I/O operations should be consistent and uniform. You print a file in the same manner as you play an audio sample, i.e., you send the device in question a WRITE command and the address of the buffer holding the data you wish to write.

The result is that the interface presented to the programmer is essentially device independent and accessible from any computer language. This greatly expands the power the Amiga brings to the programmer and, ultimately, to the user.

Devices support two types of commands: Exec standard commands like READ and WRITE, and device specific commands like the trackdisk device MOTOR command which controls the floppy drive motor, and the keyboard device READMATRIX command which returns the state of each key on the keyboard. You should keep in mind, however, that supporting standard commands does not mean that all devices execute them in *exactly* the same manner.

This chapter contains an introduction to the Exec and *amiga.lib* functions that are used when accessing Amiga devices. Consult the *Amiga ROM Kernel Manual: Devices* volume for chapters on each of the Amiga devices and the commands they support. In addition, the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* contains Autodocs summarizing the commands of each device, and listings of the device include files. Both are very useful manuals to have around when you are programming the devices.

Accessing a Device

Accessing a device requires obtaining a message port, allocating memory for a specialized message packet called an I/O request, setting a pointer to the message port in the I/O request, and finally, establishing the link to the device itself by opening it. An example of how to do this will be provided later in this chapter.

CREATING A MESSAGE PORT

When a device completes the command in a message, it will return the message to the message port specifed as the *reply* port in the message. A message port is obtained by calling the **CreateMsgPort()** or **CreatePort()** function. You must delete the message port when you are finished by calling the **DeleteMsgPort()** or **DeletePort()** function.

If your application needs to be compatible with pre-V36 versions of the operating system, use the *amiga.lib* functions **CreatePort()** and **DeletePort()**; if you require V36 or higher, you may use the Exec ROM functions **CreateMsgPort()** and **DeleteMsgPort()**.

CREATING AN IO REQUEST

The I/O request is used to send commands and data from your application to the device. The I/O request consists of fields used to hold the command you wish to execute and any parameters it requires. You set up the fields with the appropriate information and send it to the device by using Exec I/O functions. Different Amiga devices often require different I/O request structures. These structures all start with a simple **IORequest** or **IoStdReq** structure (see *<exec/io.h>*) which may be followed by various device-specific fields. Consult the Autodoc and include file for each device to determine the type and size I/O request required to access the device.

I/O request structures are commonly created and deleted with the *amiga.lib* functions **CreateExtIO()** with **DeleteExtIO()**. These *amiga.lib* functions are compatible with Release 2 and previous versions of the operating system. Applications that already require V37 for other reasons may instead use the new V37 ROM Exec functions **CreateIORequest()** and **DeleteIORequest()**. Any size and type of I/O request may be created with these functions.

Alternately, I/O requests can be created by declaring them as structures initialized to zero, or by dynamically allocating cleared public memory for them, but in these cases you will be responsible for the IORequest structure initializations which are normally handled by the above functions. The message port pointer in the I/O request tells the device where to respond with messages for your application. You must set a pointer to the message port in the I/O request if you declare it as a structure or allocate memory for it using **AllocMem()**.

OPENING A DEVICE

The device is opened by calling the **OpenDevice()** function. In addition to establishing the link to the device, **OpenDevice()** also initializes fields in the I/O request. **OpenDevice()** has this format:

```
return = OpenDevice(device_name,unit_number,(struct IORequest *)IORequest,flags)
```

- **device_name** is one of the following NULL-terminated strings for system devices:

audio.device	parallel.device	clipboard.device
printer.device	console.device	scsi.device
gameport.device	serial.device	input.device
timer.device	keyboard.device	trackdisk.device
	narrator.device	

- **unit_number** is refers to one of the logical units of the device. Devices with one unit always use unit 0. Multiple unit devices like the trackdisk device and the timer device use the different units for specific purposes.
- **IORequest** is the structure discussed above. Some of the devices have their own I/O requests defined in their include files and others use standard I/O requests, (**IOStdReq**). Refer to the *Amiga ROM Kernel Reference Manual: Devices* for more information.
- **flags** are bits set to indicate options for some of the devices. This field is set to zero for devices which don't accept options when they are opened. The flags for each device are explained in the *Amiga ROM Kernel Reference Manual: Devices.*
- **return** is an indication of whether the **OpenDevice()** was successful with zero indicating success. Never assume that a device will successfully open. Check the return value and act accordingly.

> *Zero Equals Success for OpenDevice().* Unlike most Amiga system functions, **OpenDevice()** returns zero for success and a device-specific error value for failure.

Using A Device

Once a device has been opened, you use it by passing the I/O request to it. When the device processes the I/O request, it acts on the information the I/O request contains and returns a reply message, i.e., the I/O request, to the message port when it is finished. The I/O request is passed to a device using one of three functions, **DoIO()**, **SendIO()** and **BeginIO()**. They take only one argument: the I/O request you wish to pass to the device.

- **DoIO()** is a synchronous function. It will not return until the device has finished with the I/O request. **DoIO()** will wait, if necessary, for the request to complete, and will remove (**GetMsg()**) any reply message from the message port.

- **SendIO()** is an asynchronous function. It can return immediately, but the I/O operation it initiates may take a short or long time. **SendIO** is normally used when your application has other work to do while the I/O request is being acted upon, or if your application wishes to allow the user to cancel the I/O. Using **SendIO()** requires that you wait on or check for completion of the request, and remove the completed request from the message port with **GetMsg()**.
- **BeginIO()** is commonly used to control the QuickIO bit when sending an I/O request to a device. When the QuickIO flag (IOF_QUICK) is set in the I/O request, a device is allowed to take certain shortcuts in performing and completing a request. If the request can complete immediately, the device will not return a reply message and the QuickIO flag will remain set. If the request cannot be completed immediately, the QUICK_IO flag will be clear. **DoIO()** normally requests QuickIO; **SendIO()** does not.

An I/O request typically has three fields set for every command sent to a device. You set the command itself in the **io_Command** field, a pointer to the data for the command in the **io_Data** field, and the length of the data in the **io_Length** field.

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data     = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
SendIO((struct IORequest *)SerialIO);
```

Commands consist of two parts (separated by an underscore, all in upper case): a prefix and the command word. The prefix indicates whether the command is an Exec or device specific command. All Exec standard commands have "CMD" as the prefix. They are defined in the include file *<exec/io.h>*.

Table 19-2: Standard Exec Device Commands

CMD_READ	CMD_START	CMD_UPDATE	CMD_CLEAR
CMD_WRITE	CMD_STOP	CMD_FLUSH	CMD_RESET

You should not assume that a device supports all standard Exec device commands. Always check the documentation before attempting to use one of them. Device-specific command prefixes vary with the device.

Table 19-3: System Device Command Prefixes

Device	Prefix	Example
Audio	ADCMD	ADCMD_ALLOCATE
Clipboard	CBD	CBD_POST
Console	CD	CD_ASKKEYMAP
Gameport	GPD	GPD_SETCTYPE
Input	IND	IND_SETMPORT
Keyboard	KBD	KBD_READMATRIX
Narrator	no device specific commands	-
Parallel	PDCMD	PDCMD_QUERY
Printer	PRD	PRD_PRTCOMMAND
SCSI	HD	HD_SCSICMD
Serial	SDCMD	SDCMD_BREAK
Timer	TR	TR_ADDREQUEST
Trackdisk	TD and ETD	TD_MOTOR/ETD_MOTOR

Each device maintains its own I/O request queue. When a device receives an I/O request, it either processes the request immediately or puts it in the queue because one is already being processed. After an I/O request is completely processed, the device checks its queue and if it finds another I/O request, begins to process that request.

SYNCHRONOUS VS. ASYNCHRONOUS REQUESTS

As stated above, you can send I/O requests to a device synchronously or asynchronously. The choice of which to use is largely a function of your application.

Synchronous requests use the **DoIO()** function. **DoIO()** will not return control to your application until the I/O request has been satisfied by the device. The advantage of this is that you don't have to monitor the message port for the device reply because **DoIO()** takes care of all the message handling. The disadvantage is that your application will be tied up while the I/O request is being processed, and should the request not complete for some reason, **DoIO()** will not return and your application will hang.

Asynchronous requests use the **SendIO()** and **BeginIO()** functions. Both return to your application almost immediately after you call them. This allows you to do other operations, including sending more I/O requests to the device. Note that any additional I/O requests you send must use separate I/O request structures. Outstanding I/O requests are not available for re-use until the device is finished with them.

> *Do Not Touch!* When you use **SendIO()** or **BeginIO()**, the I/O request you pass to the device and any associated data buffers should be considered read-only. Once you send it to the device, you must *not* modify it in any way until you receive the reply message from the device or abort the request.

Sending multiple asynchronous I/O requests to a device can be tricky because devices require them to be unique and initialized. This means you can't use an I/O request that's still in the queue, but you need the fields which were initialized in it when you opened the device. The solution is to copy the initialized I/O request to another I/O request(s) before sending anything to the device.

Regardless of what you do while you are waiting for an asynchronous I/O request to return, you need to have some mechanism for knowing when the request has been done. There are two basic methods for doing this.

The first involves putting your application into a wait state until the device returns the I/O request to the message port of your application. You can use the **WaitIO()**, **Wait()** or **WaitPort()** function to wait for the return of the I/O request. It is important to note that all of the above functions and also **DoIO()** may **Wait()** on the message reply port's **mp_SigBit**. For this reason, the task that created the port must be the same task the waits for completion of the I/O. There are three ways to wait:

- **WaitIO()** not only waits for the return of the I/O request, it also takes care of all the message handling functions. This is very convenient, but you can pay for this convenience: your application will hang if the I/O request does not return.

- **Wait()** waits for a signal to be sent to the message port. It will awaken your task when the signal arrives, but you are responsible for all of the message handling.

- **WaitPort()** waits for the message port to be non-empty. It returns a pointer to the message in the port, but you are responsible for all of the message handling.

The second method to detect when the request is complete involves using the **CheckIO()** function. **CheckIO()** takes an I/O request as its argument and returns an indication of whether or not it has been completed. When **CheckIO()** returns the completed indication, you will still have to remove the I/O request from the message port.

I/O REQUEST COMPLETION

A device will set the **io_Error** field of the I/O request to indicate the success or failure of an operation. The indication will be either zero for success or a non-zero error code for failure. There are two types of error codes: Exec I/O and device specific. Exec I/O errors are defined in the include file *<exec/errors.h>*; device specific errors are defined in the include file for each device. You should always check that the operation you requested was successful.

The exact method for checking **io_Error** can depend on whether you use **DoIO()** or **SendIO()**. In both cases, **io_Error** will be set when the I/O request is returned, but in the case of **DoIO()**, the **DoIO()** function itself returns the same value as **io_Error**. This gives you the option of checking the function return value:

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data     = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
if (DoIO((struct IORequest *)SerialIO)
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Or you can check **io_Error** directly:

```
SerialIO->IOSer.io_Length   = sizeof(ReadBuffer);
SerialIO->IOSer.io_Data     = ReadBuffer;
SerialIO->IOSer.io_Command  = CMD_READ;
DoIO((struct IORequest *)SerialIO);
if (SerialIO->IOSer.io_Error)
    printf("Read failed.  Error: %ld\n",SerialIO->IOSer.io_Error);
```

Keep in mind that checking **io_Error** is the *only* way that I/O requests sent by **SendIO()** can be checked. Testing for a failed I/O request is a minimum step, what you do beyond that depends on your application. In some instances, you may decide to resend the I/O request and in others, you may decide to stop your application. One thing you'll almost always want to do is to inform the user that an error has occurred.

> *Exiting The Correct Way.* If you decide that you must prematurely end your application, you should deallocate, release, give back and let go of everything you took to run the application. In other words, you should exit gracefully.

Closing the Device

You end device access by reversing the steps you did to access it. This means you close the device, deallocate the I/O request memory and delete the message port. In that order!

Closing a device is how you tell Exec that you are finished using a device and any associated resources. This can result in housecleaning being performed by the device. However, before you close a device, you might have to do some housecleaning of your own.

A device is closed by calling the **CloseDevice()** function. The **CloseDevice()** function does not return a value. It has this format:

```
CloseDevice(IORequest);
```

where **IORequest** is the I/O request used to open the device.

You should not close a device while there are outstanding I/O requests, otherwise you can cause major and minor problems. Let's begin with the minor problem: memory. If an I/O request is outstanding at the time you close a device, you won't be able to reclaim the memory you allocated for it.

The major problem: the device will try to respond to the I/O request. If the device tries to respond to an I/O request, and you've deleted the message port (which is covered below), you will probably crash the system.

One solution would be to wait until all I/O requests you sent to the device return. This is not always practical if you've sent a few requests and the user wants to exit the application immediately.

In that case, the only solution is to abort and remove any outstanding I/O requests. You do this with the functions **AbortIO()** and **WaitIO()**. They must be used together for cleaning up. **AbortIO()** will abort an I/O request, but will not prevent a reply message from being sent to the application requesting the abort. **WaitIO()** will wait for an I/O request to complete and remove it from the message port. This is why they must be used together.

> *Be Careful With AbortIO()!* Do not **AbortIO()** an I/O request which has *not* been sent to a device. If you do, you may crash the system.

Ending Device Access

After the device is closed, you must deallocate the I/O request memory. The exact method you use depends on how you allocated the memory in the first place. For **AllocMem()** you call **FreeMem()**, for **CreateExtIO()** you call **DeleteExtIO()**, and for **CreateIORequest()** you call **DeleteIORequest()**. If you allocated the I/O request memory at compile time, you naturally have nothing to free.

Finally, you must delete the message port you created. You delete the message port by calling **DeleteMsgPort()** if you used **CreateMsgPort()**, or **DeletePort()** if you used **CreatePort()**.

Here is the checklist for gracefully exiting:

- Abort any outstanding I/O requests with **AbortIO()**.
- Wait for the completion of any outstanding or aborted I/O requests with **WaitIO()**.
- Close the device with **CloseDevice()**.
- Release the I/O request memory with either **DeleteIORequest()**, **DeleteExtIO()** or **FreeMem()** (as appropriate).
- Delete the message port with **DeleteMsgPort()** or **DeletePort()**.

Devices With Functions

Some devices, in addition to their commands, provide library-style functions which can be directly called by applications. These functions are documented in the device specific FD files and Autodocs of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*, and in the *Devices* volume of this manual set.

Devices with functions behave much like Amiga libraries, i.e., you set up a base address pointer and call the functions as offsets from the pointer. See the "Exec Libraries" chapter for more information.

The procedure for accessing a device's functions is as follows:

- Declare the device base address variable in the global data area. The correct name for the base address can be found in the device's FD file.
- Create a message port data structure.
- Create an I/O request data structure.
- Call **OpenDevice()**, passing the I/O request. If **OpenDevice()** is successful (returns 0), the address of the device base may be found in the **io_Device** field of the I/O request structure. Consult the include file for the structure you are using to determine the full name of the **io_Device** field. The base address is only valid while the device is open.
- Set the device base address variable to the pointer returned in the **io_Device** field.

We will use the timer device to illustrate the above method. The name of the timer device base address is listed in its FD file as **TimerBase**.

```
#include <devices/timer.h>

struct Library *TimerBase;              /* device base address pointer */
struct MsgPort *TimerMP;                /* message port pointer       */
struct timerequest *TimerIO;            /* I/O request pointer        */

if (TimerMP=CreatePort(NULL,NULL))      /* Create the message port.   */
{
    /* Create the I/O request.  */
    if ( TimerIO = (struct timerequest *)
         CreateExtIO(TimerMP,sizeof(struct timerequest)) )
    {
        /* Open the timer device.  */
        if ( !(OpenDevice(TIMERNAME,UNIT_MICROHZ,TimerIO,0)) )
        {
            /* Set up pointer for timer functions.  */
            TimerBase = (struct Library *)TimerIO->tr_node.io_Device;

            /* Use timer device library-style functions such as CmpTime() ...*/

            CloseDevice(TimerIO);              /* Close the timer device. */
        }
        else
            printf("Error: Could not open %s\n",TIMERNAME);
    }
    else
        printf("Error: Could not create I/O request\n");
}
else
    printf("Error: Could not create message port\n");
}
```

Using An Exec Device

The following short example demonstrates use of an Amiga device. The example opens the serial.device and then demonstrates both synchronous (**DoIO()**) and asynchronous (**SendIO()**) use of the serial command SDCMD_QUERY. This command is used to determine the status of the serial device lines and registers. The example uses the backward compatible *amiga.lib* functions for creation and deletion of the message port and I/O request.

```
;/* DeviceUse.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 DeviceUse.c
Blink FROM LIB:c.o,DeviceUse.o TO DeviceUse LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

/* DeviceUse.c - an example of using an Amiga device (here, serial device)    */
/*    - attempt to create a message port with CreatePort()   (from amiga.lib) */
/*    - attempt to create the I/O request with CreateExtIO() (from amiga.lib) */
/*    - attempt to open the serial device with Exec OpenDevice()              */
/*                                                                            */
/* If successful, use the serial command SDCMD_QUERY, then reverse our steps. */
/* If we encounter an error at any time, we will gracefully exit.  Note that  */
/* applications which require at least V37 OS should use the Exec functions   */
/* CreateMsgPort()/DeleteMsgPort() and CreateIORequest()/DeleteIORequest()    */
/* instead of the similar amiga.lib functions which are used in this example. */

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>

#include <clib/exec_protos.h> /* Prototypes for Exec library functions */
#include <clib/alib_protos.h> /* Prototypes for amiga.lib functions    */

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable SAS CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

void main(void)
{
    struct MsgPort  *serialMP;      /* for pointer to our message port */
    struct IOExtSer *serialIO;      /* for pointer to our I/O request  */
    struct IOExtSer *reply;         /* for use with GetMsg             */

    if (serialMP=CreatePort(NULL,NULL)) /* Create the message port. */
    {
        /* Create the I/O request. Note that <devices/serial.h> defines the type */
        /* of IORequest required by the serial device--an IOExtSer. Many devices */
        /* require specialized extended IO requests which start with an embedded */
        /* struct IORequest. The generic Exec and amiga.lib device IO functions  */
        /* are prototyped for IORequest, so some pointer casting is necessary.   */

        if (serialIO = (struct IOExtSer *)CreateExtIO(serialMP,sizeof(struct IOExtSer)))
        {
            /* Open the serial device (non-zero return value means failure here). */
            if (OpenDevice( SERIALNAME, 0, (struct IORequest *)serialIO, 0L))
                printf("Error: %s did not open\n",SERIALNAME);
            else
            {
                /* Device is open */                         /* DoIO - demonstrates synchronous */
                serialIO->IOSer.io_Command  = SDCMD_QUERY;   /* device use, returns error or 0. */
                if (DoIO((struct IORequest *)serialIO))
                    printf("Query  failed. Error - %d\n",serialIO->IOSer.io_Error);
                else
                    /* Print serial device status - see include file for meaning */
                    /* Note that with DoIO, the Wait and GetMsg are done by Exec */
                    printf("Serial device status: $%x\n\n",serialIO->io_Status);

                serialIO->IOSer.io_Command  = SDCMD_QUERY; /* SendIO - demonstrates asynchronous */
                SendIO((struct IORequest *)serialIO);      /* device use (returns immediately).  */
```

```
            /* We could do other things here while the query is being done.      */
            /* And to manage our asynchronous device IO:                           */
            /*   - we can CheckIO(serialIO) to check for completion                */
            /*   - we can AbortIO(serialIO) to abort the command                   */
            /*   - we can WaitPort(serialMP) to wait for any serial port reply     */
            /*  OR we can WaitIO(serialIO) to wait for this specific IO request    */
            /*  OR we can Wait(1L << serialMP_>mp_SigBit) for reply port signal    */

            Wait(1L << serialMP->mp_SigBit);

            while(reply = (struct IOExtSer *)GetMsg(serialMP))
            {    /* Since we sent out only one serialIO request the while loop is */
                 /* not really needed--we only expect one reply to our one query  */
                 /* command, and the reply message pointer returned by GetMsg()   */
                 /* will just be another pointer to our one serialIO request.     */
                 /* With Wait() or WaitPort(), you must GetMsg() the message.     */
                if(reply->IOSer.io_Error)
                    printf("Query  failed. Error - %d\n",reply->IOSer.io_Error);
                else
                    printf("Serial device status: $%x\n\n",reply->io_Status);
            }
            CloseDevice((struct IORequest *)serialIO);  /* Close the serial device.    */
        }
        DeleteExtIO(serialIO);                          /* Delete the I/O request.     */
    }
    else printf("Error: Could create I/O request\n");   /* Inform use. that the I/O    */
                                                        /* request could be created.   */
    DeletePort(serialMP);                               /* Delete the message port.    */
}
else printf("Error: Could not create message port\n");  /* Inform user that the message*/
}                                                       /* port could not be created.  */
```

Function Reference

The following chart gives a brief description of the Exec functions that control device I/O. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 19-4: Exec Device I/O Functions

Exec Device I/O Function	Description
CreateIORequest()	Create an **IORequest** structure (V36).
DeleteIORequest()	Delete an **IORequest** created by **CreateIORequest()** (V36).
OpenDevice()	Gain access to an Exec device.
CloseDevice()	Close Exec device opened with **OpenDevice()**.
DoIO()	Perform a device I/O command and wait for completion.
SendIO()	Initiate an I/O command. Do not wait for it to complete.
CheckIO()	Get the status of an **IORequest**.
WaitIO()	Wait for completion of an I/O request.
AbortIO()	Attempt to abort an I/O request that is in progress.

Table 19-5: Exec Support Functions in amiga.lib

Function	Description
BeginIO()	Initiate an asynchronous device I/O request.
CreateExtIO()	Create an **IORequest** data structure.
DeleteExtIO()	Free an **IORequest** structure allocated by **CreateExtIO()**.

Chapter 20
EXEC MEMORY ALLOCATION

Exec manages all of the free memory currently available in the system. Using linked list structures, Exec keeps track of memory and provides the functions to allocate and access it.

When an application needs some memory, it can either declare the memory statically within the program or it can ask Exec for some memory. When Exec receives a request for memory, it looks through its list of free memory regions to find a suitably sized block that matches the size and attributes requested.

Memory Functions

Normally, an application uses the **AllocMem()** function to ask for memory:

```
APTR AllocMem(ULONG byteSize, ULONG attributes);
```

The **byteSize** argument is the amount of memory the application needs and **attributes** is a bit field which specifies any special memory characteristics (described later). If **AllocMem()** is successful, it returns a pointer to a block of memory. The memory allocation will fail if the system cannot find a big enough block with the requested attributes. If **AllocMem()** fails, it returns NULL.

Because the system only keeps track of how much free memory is available and not how much is in use, it has no idea what memory has been allocated by any task. This means an application has to explicitly return, or deallocate, any memory it has allocated so the system can return that memory to the free memory list. If an application does not return a block of memory to the system, the system will not be able to reallocate that memory to some other task. That block of memory will be lost until the Amiga is reset. If you are using **AllocMem()** to allocate memory, a call to **FreeMem()** will return that memory to the system:

```
void FreeMem(APTR mymemblock, ULONG byteSize);
```

Here **mymemblock** is a pointer to the memory block the application is returning to the system and **byteSize** is the same size that was passed when the memory was allocated with **AllocMem()**.

Unlike some compiler memory allocation functions, the Amiga system memory allocation functions return memory blocks that are at least longword aligned. This means that the allocated memory will always start on an address which is at least evenly divisible by four. This alignment makes the memory suitable for any system structures or buffers which require word or longword alignment, and also provides optimal alignment for stacks and memory copying.

MEMORY ATTRIBUTES

When asking the system for memory, an application can ask for memory with certain attributes. The currently supported flags are listed below. Flags marked "V37" are new memory attributes for Release 2. Allocations which specify these new bits may fail on earlier systems.

MEMF_ANY
This indicates that there is no requirement for either Fast or Chip memory. In this case, while there is Fast memory available, Exec will only allocate Fast memory. Exec will allocate Chip memory if there is not enough Fast memory.

MEMF_CHIP
This indicates the application wants a block of chip memory, meaning it wants memory addressable by the Amiga custom chips. Chip memory is required for any data that will be accessed by custom chip DMA. This includes screen memory, images that will be blitted, sprite data, copper lists, and audio data, and pre-V37 floppy disk buffers. If this flag is not specified when allocating memory for these types of data, your code will fail on machines with expanded memory.

MEMF_FAST
This indicates a memory block outside of the range that the special purpose chips can access. "FAST" means that the special-purpose chips do not have access to the memory and thus cannot cause processor bus contention, therefore processor access will likely be faster. Since the flag specifies memory that the custom chips cannot access, this flag is mutually exclusive with the MEMF_CHIP flag. If you specify the MEMF_FAST flag, your allocation will fail on any Amiga that has only CHIP memory. Use MEMF_ANY if you would *prefer* FAST memory.

MEMF_PUBLIC
This indicates that the memory should be accessible to other tasks. Although this flag doesn't do anything right now, using this flag will help ensure compatibility with possible future features of the OS (like virtual memory and memory protection).

MEMF_CLEAR
This indicates that the memory should be initialized with zeros.

MEMF_LOCAL (V37)
This indicates memory which is located on the motherboard which is not initialized on reset.

MEMF_24BITDMA (V37)
This indicates that the memory should be allocated within the 24 bit address space, so that the memory can be used in Zorro-II expansion device DMA transactions. This bit is for use by Zorro-II DMA devices only. It is not for general use by applications.

MEMF_REVERSE (V37)
Indicates that the memory list should be searched backwards for the highest address memory chunk which can be used for the memory allocation.

If an application does not specify any attributes when allocating memory, the system tries to satisfy the request with the first memory available on the system memory lists, which is MEMF_FAST if available, followed by MEMF_CHIP.

> *Make Sure You Have Memory.* Always check the result of any memory allocation to be sure the type and amount of memory requested is available. Failure to do so will lead to trying to use an non-valid pointer.

ALLOCATING SYSTEM MEMORY

The following examples show how to allocate memory.

```
APTR  apointer,anotherptr, yap;

if (!(apointer = AllocMem(100, MEMF_ANY)))
    {  /* COULDN'T GET MEMORY, EXIT */ }
```

AllocMem() returns the address of the first byte of a memory block that is at least 100 bytes in size or NULL if there is not that much free memory. Because the requirement field is specified as MEMF_ANY (zero), memory will be allocated from any one of the system-managed memory regions.

```
if (!(anotherptr = (APTR)AllocMem(1000, MEMF_CHIP | MEMF_CLEAR)))
    {  /* COULDN'T GET MEMORY, EXIT */ }
```

The example above allocates only chip-accessible memory, which the system fills with zeros before it lets the application use the memory. If the system free memory list does not contain enough contiguous memory bytes in an area matching your requirements, **AllocMem()** returns a zero. You *must* check for this condition.

If you are using Release 2, you can use the **AllocVec()** function to allocate memory. In addition to allocating a block of memory, this function keeps track of the size of the memory block, so your application doesn't have to remember it when it deallocates that memory block. The **AllocVec()** function allocates a little more memory to store the size of the memory allocation request.

```
if (!(yap = (APTR)AllocVec(512, MEMF_CLEAR)))
    {  /* COULDN'T GET MEMORY, EXIT */ }
```

FREEING SYSTEM MEMORY

The following examples free the memory chunks shown in the previous calls to **AllocMem()**.

```
FreeMem(apointer, 100);
FreeMem(anotherptr, 1000);
```

A memory block allocated with **AllocVec()** must be returned to the system pool with the **FreeVec()**. This function uses the stored size in the allocation to free the memory block, so there is no need to specify the size of the memory block to free.

```
FreeVec(yap);
```

FreeMem() and **FreeVec()** return no status. However, if you attempt to free a memory block in the middle of a chunk that the system believes is already free, you will cause a system crash. Applications must free the same size memory blocks that they allocated. An allocated block may not be deallocated as smaller pieces. Due to the internal way the system rounds up and aligns allocations. Partial deallocations can corrupt the system memory list.

> *Leave Memory Allocations Out Of Interrupt Code.* Do not allocate or deallocate system memory from within interrupt code. The "Exec Interrupts" chapter explains that an interrupt may occur at any time, even during a memory allocation process. As a result, system data structures may not be internally consistent at this time.

MEMORY INFORMATION FUNCTIONS

The memory information routines **AvailMem()** and **TypeOfMem()** can provide the amount of memory available in the system, and the attributes of a particular block of memory.

Memory Requirements

The same attribute flags used in memory allocation routines are valid for the memory information routines. There is also an additional flag, MEMF_LARGEST, which can be used in the **AvailMem()** routine to find out what the largest available memory block of a particular type is. Specifying the MEMF_TOTAL flag will return the total amount of memory currently available.

Calling Memory Information Functions

The following example shows how to find out how much memory of a particular type is available.

```
ULONG size;

size = AvailMem(MEMF_CHIP|MEMF_LARGEST);
```

AvailMem() returns the size of the largest chunk of available chip memory.

> *AvailMem() May Not Be Totally Accurate.* Because of multitasking, the return value from **AvailMem()** may be inaccurate by the time you receive it.

The following example shows how to determine the type of memory of a specified memory address.

```
ULONG memtype;

memtype = TypeOfMem((APTR)0x090000);
if ((memtype & MEMF_CHIP) == MEMF_CHIP)  {  /*   . . . It's chip memory . . .   */   }
```

TypeOfMem() returns the attributes of the memory at a specific address. If it is passed an invalid memory address, **TypeOfMem()** returns NULL. This routine is normally used to determine if a particular chunk of memory is in chip memory.

USING MEMORY COPY FUNCTIONS

For memory block copies, the **CopyMem()** and **CopyMemQuick()** functions can be used.

Copying System Memory

The following samples show how to use the copying routines.

```
APTR source, target;

source = AllocMem(1000, MEMF_CLEAR);
target = AllocMem(1000, MEMF_CHIP);
CopyMem(source, target, 1000);
```

CopyMem() copies the specified number of bytes from the source data region to the target data region. The pointers to the regions can be aligned on arbitrary address boundaries. **CopyMem()** will attempt to copy the memory as efficiently as it can according to the alignment of the memory blocks, and the amount of data that it has to transfer. These functions are optimized for copying large blocks of memory which can result in unnecessary overhead if used to transfer very small blocks of memory.

```
CopyMemQuick(source, target, 1000);
```

CopyMemQuick() performs an optimized copy of the specified number of bytes from the source data region to the target data region. The source and target pointers must be longword aligned and the size (in bytes) must be divisible by four.

> *Not All Copies Are Supported.* Neither **CopyMem()** nor **CopyMemQuick()** supports copying between regions that overlap.

SUMMARY OF SYSTEM CONTROLLED MEMORY HANDLING ROUTINES

AllocMem() and **FreeMem()**

These are system-wide memory allocation and deallocation routines. They use a memory free-list owned and managed by the system.

AvailMem()

This routine returns the number of free bytes in a specified type of memory.

TypeOfMem()

This routine returns the memory attributes of a specified memory address.

CopyMem()/CopyMemQuick()

CopyMem() is a general purpose memory copy routine. **CopyMemQuick()** is an optimized version of **CopyMemQuick()**, but has restrictions on the size and alignment of the arguments.

Allocating Multiple Memory Blocks

Exec provides the routines **AllocEntry()** and **FreeEntry()** to allocate multiple memory blocks in a single call. **AllocEntry()** accepts a data structure called a **MemList**, which contains the information about the size of the memory blocks to be allocated and the requirements, if any, that you have regarding the allocation. The **MemList** structure is found in the include file *<exec/memory.h>* and is defined as follows:

```
struct MemList
{
    struct Node     ml_Node;
    UWORD           ml_NumEntries;      /* number of MemEntrys */
    struct MemEntry ml_ME[1];           /* where the MemEntrys begin*/
};
```

Node

allows you to link together multiple **MemLists**. However, the node is ignored by the routines **AllocEntry()** and **FreeEntry()**.

ml_NumEntries

tells the system how many **MemEntry** sets are contained in this **MemList**. Notice that a **MemList** is a variable-length structure and can contain as many sets of entries as you wish.

The **MemEntry** structure looks like this:

```
struct MemEntry
{
    union {
        ULONG   meu_Reqs;    /* the AllocMem requirements */
        APTR    meu_Addr;    /* address of your memory */
        } me_Un;
    ULONG   me_Length;       /* the size of this request */
};
```

SAMPLE CODE FOR ALLOCATING MULTIPLE MEMORY BLOCKS

Here's an example of showing how to use the **AllocEntry()** with multiple blocks of memory.

```
;/* allocentry.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 allocentry.c
Blink FROM LIB:c.o,allocentry.o TO allocentry LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;

allocentry.c - example of allocating several memory areas.
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <clib/exec_protos.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef LATTICE
int CXBRK(void)   { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }   /* really */
#endif

#define ALLOCERROR 0x80000000

struct MemList *memlist;             /* pointer to a MemList structure       */

struct MemBlocks /* define a new structure because C cannot initialize unions */
{
    struct MemList  mn_head;         /* one entry in the header              */
    struct MemEntry mn_body[3];      /* additional entries follow directly as */
} memblocks;                         /* part of the same data structure      */

VOID main(VOID)
{
    memblocks.mn_head.ml_NumEntries = 4; /* 4! Since the MemEntry starts at 1! */

    /* Describe the first piece of memory we want.  Because of our MemBlocks structure */
    /* setup, we reference the first MemEntry differently when initializing it.       */
    memblocks.mn_head.ml_ME[0].me_Reqs = MEMF_CLEAR;
    memblocks.mn_head.ml_ME[0].me_Length = 4000;

    memblocks.mn_body[0].me_Reqs   = MEMF_CHIP | MEMF_CLEAR;   /* Describe the other pieces of    */
    memblocks.mn_body[0].me_Length = 100000;                   /* memory we want. Additional      */
    memblocks.mn_body[1].me_Reqs   = MEMF_PUBLIC | MEMF_CLEAR; /* MemEntries are initialized this */
    memblocks.mn_body[1].me_Length = 200000;                   /* way. If we wanted even more en- */
    memblocks.mn_body[2].me_Reqs   = MEMF_PUBLIC;              /* tries, we would need to declare */
    memblocks.mn_body[2].me_Length = 25000;                    /* a larger MemEntry array in our  */
                                                               /* MemBlocks structure.            */

    memlist = (struct MemList *)AllocEntry((struct MemList *)&memblocks);

    if ((ULONG)memlist & ALLOCERROR)         /* 'error' bit 31 is set (see below). */
    {
       printf("AllocEntry FAILED\n");
       exit(200);
    }
    /* We got all memory we wanted.  Use it and call FreeEntry() to free it */
    printf("AllocEntry succeeded - now freeing all allocated blocks\n");
    FreeEntry(memlist);
}
```

AllocEntry() returns a pointer to a new **MemList** of the same size as the **MemList** that you passed to it. For example, ROM code can provide a **MemList** containing the requirements of a task and create a RAM-resident copy of the list containing the addresses of the allocated entries. The pointer to the **MemList** is used as the argument for **FreeEntry()** to free the memory blocks.

> *Assembly Does Not Have MemEntry.* The **MemList** structure used by assembly programmers is slightly different; it has only a label for the start of the **MemEntry** array. See the Exec **AllocEntry()** Autodoc for an example of using **AllocEntry()** from assembler.

RESULT OF ALLOCATING MULTIPLE MEMORY BLOCKS

The **MemList** created by **AllocEntry()** contains **MemEntry** entries. **MemEntrys** are defined by a union statement, which allows one memory space to be defined in more than one way.

If **AllocEntry()** returns a value with bit 31 clear, then all of the **meu_Addr** positions in the returned **MemList** will contain valid memory addresses meeting the requirements you have provided. To use this memory area, you would use code similar to the following:

```
#define ALLOCERROR 0x80000000
struct  MemList *ml;
APTR    data, moredata;

if ( ! ((ULONG)ml & ALLOCERROR))    /* After calling AllocEntry to allocate ml */
{
    data     = ml->ml_ME[0].me_Addr;
    moredata = ml->ml_ME[1].me_Addr;
}
else  exit(200);                    /* error during AllocEntry */
```

If **AllocEntry()** has problems while trying to allocate the memory you have requested, instead of the address of a new **MemList**, it will return the memory requirements value with which it had the problem. Bit 31 of the value returned will be set, and no memory will be allocated. Entries in the list that were already allocated will be freed. For example, a failed allocation of cleared Chip memory (MEMF_CLEAR | MEMF_CHIP) could be indicated with 0x80010002, where bit 31 indicates failure, bit 16 is the MEMF_CLEAR flag and bit 1 is the MEMF_CHIP flag.

MULTIPLE MEMORY BLOCKS AND TASKS

If you want to take advantage of Exec's automatic cleanup, use the **MemList** and **AllocEntry()** facility to do your dynamic memory allocation.

In the **Task** control block structure, there is a list header named **tc_MemEntry**. This is the list header that you initialize to include **MemLists** that your task has created by call(s) to **AllocEntry()**. Here is a short program segment that handles task memory list header initialization only. It assumes that you have already run **AllocEntry()** as shown in the simple **AllocEntry()** example above.

```
struct Task *tc;
struct MemList *ml;

/* First initialize the task pointer and AllocEntry() the memlist ml */

if(!tc->tc_MemEntry)
    NewList(tc->tc_MemEntry);  /* Initialize the task's memory    */
                               /* list header. Do this once only! */
AddTail(tc->tc_MemEntry, ml);
```

Assuming that you have only used the **AllocEntry()** method (or **AllocMem()** and built your own custom **MemList**), the system now knows where to find the blocks of memory that your task has dynamically allocated. The **RemTask()** function automatically frees all memory found on **tc_MemEntry**.

> *CreateTask() Sets Up A MemList.* The *amiga.lib* **CreateTask()** function, and other system task and process creation functions use a **MemList** in **tc_MemEntry** so that the Task structure and stack will be automatically deallocated when the Task is removed.

SUMMARY OF MULTIPLE MEMORY BLOCKS ALLOCATION ROUTINES

AllocEntry() and **FreeEntry()**

These are routines for allocating and freeing multiple memory blocks with a single call.

InitStruct()

This routine initializes memory from data and offset values in a table. Typically only assembly language programs benefit from using this routine. See the *Amiga ROM Kernel Reference Manual: Include & Autodocs* for more details.

Other Memory Functions

Allocate() and **Deallocate()** use a memory region header, called **MemHeader**, as part of the calling sequence. You can build your own local header to manage memory locally. This structure takes the form:

```
struct MemHeader {
    struct Node       mh_Node;
    UWORD             mh_Attributes;  /* characteristics of this region */
    struct  MemChunk *mh_First;       /* first free region              */
    APTR              mh_Lower;       /* lower memory bound             */
    APTR              mh_Upper;       /* upper memory bound + 1         */
    ULONG             mh_Free;        /* total number of free bytes     */
};
```

mh_Attributes

is ignored by **Allocate()** and **Deallocate()**.

mh_First

is the pointer to the first **MemChunk** structure.

mh_Lower

is the lowest address within the memory block. This must be a multiple of eight bytes.

mh_Upper

is the highest address within the memory block + 1. The highest address will itself be a multiple of eight if the block was allocated to you by **AllocMem()**.

mh_Free

is the total free space.

This structure is included in the include files *<exec/memory.h>* and *<exec/memory.i>*.

The following sample code fragment shows the correct initialization of a **MemHeader** structure. It assumes that you wish to allocate a block of memory from the global pool and thereafter manage it yourself using **Allocate()** and **Deallocate()**.

```
;/* allocate.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 allocate.c
Blink FROM LIB:c.o,allocate.o TO allocate LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;
allocate.c - example of allocating and using a private memory pool.
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <clib/exec_protos.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef LATTICE
int CXBRK(void)  { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

#define BLOCKSIZE 4000     /* or whatever you need */

VOID main(VOID)
{
    struct MemHeader *mh;
    struct MemChunk  *mc;
    APTR   block1, block2;

    /* Get the MemHeader needed to keep track of our new block. */
    mh = (struct MemHeader *)AllocMem((LONG)sizeof(struct MemHeader), MEMF_CLEAR);
    if (!mh) exit(10);

    /* Get the actual block the above MemHeader will manage. */
    if ( !(mc = (struct MemChunk *)AllocMem(BLOCKSIZE, 0)) );
    {
        FreeMem(mh, (LONG)sizeof(struct MemHeader));
        exit(10);
    }
    mh->mh_Node.ln_Type = NT_MEMORY;
    mh->mh_First        = mc;
    mh->mh_Lower        = (APTR)mc;
    mh->mh_Upper        = (APTR)(BLOCKSIZE + (ULONG)mc);
    mh->mh_Free         = BLOCKSIZE;

    mc->mc_Next  = NULL;                     /* Set up first chunk in the freelist */
    mc->mc_Bytes = BLOCKSIZE;

    block1 = (APTR)Allocate(mh,20);
    block2 = (APTR)Allocate(mh, 314);

    printf("Our MemHeader struct at $%lx. Our block of memory at $%lx\n", mh, mc);
    printf("Allocated from our pool: block1 at $%lx, block2 at $%lx\n", block1, block2);

    FreeMem(mh, (LONG)sizeof(struct MemHeader));
    FreeMem(mc, (LONG)BLOCKSIZE);
}
```

> *How Memory Is Tagged.* Only free memory is "tagged" using a **MemChunk** linked list. Once memory is allocated, the system has no way of determining which task now has control of that memory.

If you allocate memory from the system, be sure to deallocate it when your task exits. You can accomplish this with matched deallocations, or by adding a **MemList** to your task's **tc_MemEntry**, or you can deallocate the memory in the **finalPC** routine (which can be specified if you perform **AddTask()** yourself).

ALLOCATING MEMORY AT AN ABSOLUTE ADDRESS

For special advanced applications, **AllocAbs()** is provided. Using **AllocAbs()**, an application can allocate a memory block starting at a specified absolute memory address. If the memory is already allocated or if there is not enough memory available for the request, **AllocAbs()** returns a zero.

Be aware that an absolute memory address which happens to be available on one Amiga may not be available on a machine with a different configuration or different operating system revision, or even on the same machine at a different times. For example, a piece of memory that is available during expansion board configuration might not be available at earlier or later times. Here is an example call to **AllocAbs()**:

```
APTR absoluteptr;

absoluteptr = (APTR)AllocAbs(10000, 0x2F0000);
if (!(absoluteptr))
    { /* Couldn't get memory, act accordingly. */  }

/* After we're done using it, we call FreeMem() to free the memory block. */
FreeMem(absoluteptr, 10000);
```

ADDING MEMORY TO THE SYSTEM POOL

When non-Autoconfig memory needs to be added to the system free pool, the **AddMemList()** function can be used. This function takes the size of the memoryblock, its type, the priority for the memory list, the base address and the name of the memory block. A **MemHeader** structure will be placed at the start of the memory block, the remainder of the memory block will be made available for allocation. For example:

```
AddMemList(0x200000, MEMF_FAST, 0, 0xF00000, "FZeroBoard");
```

will add a two megabyte memory block, starting at $F00000 to the system free pool as Fast memory. The memory list entry is identified with "FZeroBoard".

Function Reference

The following are brief descriptions of the Exec functions that handle memory management. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each call.

Table 20-1: Exec Memory Functions

Memory Function	Description
AllocMem()	Allocate memory with specified attributes. If an application needs to allocate some memory, it will usually use this function.
AddMemList()	Add memory to the system free pool.
AllocAbs()	Allocate memory at a specified location.
Allocate()	Allocate memory from a private memory pool.
AllocEntry()	Allocate multiple memory blocks.
AllocVec()	Allocate memory with specified attributes and keep track of the size (V36).
AvailMem()	Return the amount of free memory, given certain conditions.
CopyMem()	Copy memory block, which can be non-aligned and of arbitrary length.
CopyMemQuick()	Copy aligned memory block.
Deallocate()	Return memory block allocated, with **Allocate()** to the private memory pool.
FreeEntry()	Free multiple memory blocks, allocated with **AllocEntry()**.
FreeMem()	Free a memory block of specified size, allocated with **AllocMem()**.
FreeVec()	Free a memory block allocated with **AllocVec()**.
InitStruct()	Initialize memory from a table.
TypeOfMem()	Determine attributes of a specified memory address.

Chapter 21
EXEC TASKS

One of the most powerful features of the Amiga operating system is its ability to run and manage multiple independent program tasks, providing each task with processor time based on their priority and activity. These tasks include system device drivers, background utilities, and user interface environments, as well as normal application programs. This multitasking capability is provided by the Exec library's management of task creation, termination, scheduling, event signals, traps, exceptions, and mutual exclusion.

This chapter deals with Exec on a lower level than most applications programmers need and assumes you are already familiar with the Exec basics discussed in the ''Introduction to Exec'' chapter of this manual.

TASK STRUCTURE

Exec maintains task context and state information in a task-control data structure. Like most Exec structures, **Task** structures are dynamically linked onto various task queues through the use of an embedded Exec list **Node** structure (see the Lists chapter). Any task can find its own task structure by calling **FindTask**(NULL). The C-language form of this structure is defined in the <exec/tasks.h> include file:

```
struct Task  {
    struct Node tc_Node;
    UBYTE       tc_Flags;
    UBYTE       tc_State;
    BYTE        tc_IDNestCnt;   /* intr disabled nesting */
    BYTE        tc_TDNestCnt;   /* task disabled nesting */
    ULONG       tc_SigAlloc;    /* sigs allocated */
    ULONG       tc_SigWait;     /* sigs we are waiting for */
    ULONG       tc_SigRecvd;    /* sigs we have received */
    ULONG       tc_SigExcept;   /* sigs we will take excepts for */
    UWORD       tc_TrapAlloc;   /* traps allocated */
    UWORD       tc_TrapAble;    /* traps enabled */
    APTR        tc_ExceptData;  /* points to except data */
    APTR        tc_ExceptCode;  /* points to except code */
    APTR        tc_TrapData;    /* points to trap code */
    APTR        tc_TrapCode;    /* points to trap data */
    APTR        tc_SPReg;       /* stack pointer */
    APTR        tc_SPLower;     /* stack lower bound */
    APTR        tc_SPUpper;     /* stack upper bound + 2*/
    VOID      (*tc_Switch)();   /* task losing CPU */
    VOID      (*tc_Launch)();   /* task getting CPU */
    struct List tc_MemEntry;    /* allocated memory */
    APTR        tc_UserData;    /* per task data */
};
```

A similar assembly code structure is available in the <exec/tasks.i> include file.

Most of these fields are not relevant for simple tasks; they are used by Exec for state and administrative purposes. A few fields, however, are provided for the advanced programs that support higher level environments (as in the case of *processes*) or require precise control (as in *devices*). The following sections explain these fields in more detail.

Task Creation

To create a new task you must allocate a task structure, initialize its various fields, and then link it into Exec with a call to **AddTask()**. The task structure may be allocated by calling the **AllocMem()** function with the MEMF_CLEAR and MEMF_PUBLIC allocation attributes. These attributes indicate that the data structure is to be pre-initialized to zero and that the structure is shared.

The **Task** fields that require initialization depend on how you intend to use the task. For the simplest of tasks, only a few fields must be initialized:

tc_Node
: The task list node structure. This includes the task's priority, its type, and its name (refer to the "Exec Lists and Queues" chapter).

tc_SPLower
: The lower memory bound of the task's stack.

tc_SPUpper
: The upper memory bound of the task's stack.

tc_SPReg
: The initial stack pointer. Because task stacks grow *downward* in memory, this field is usually set to the same value as **tc_SPUpper**.

Zeroing all other unused fields will cause Exec to supply the appropriate system default values. Allocating the structure with the MEMF_CLEAR attribute is an easy way to be sure that this happens.

Once the structure has been initialized, it must be linked to Exec. This is done with a call to **AddTask()** in which the following parameters are specified:

```
AddTask(struct Task *task, APTR initialPC, APTR finalPC )
```

The **task** argument is a pointer to your initialized **Task** structure. Set **initialPC** to the entry point of your task code. This is the address of the first instruction the new task will execute.

Set **finalPC** to the address of the finalization code for your task. This is a code section that will receive control if the **initialPC** routine ever performs a return (RTS). This exists to prevent your task from being launched into random memory upon an accidental return. The **finalPC** routine should usually perform various program-related clean-up duties and should then remove the task. If a zero is supplied for this parameter, Exec will use its default finalization code (which simply calls the **RemTask()** function).

Under Release 2, **AddTask()** returns the address of the newly added task or NULL for failure. Under 1.3 and older versions of the OS, no values are returned.

TASK CREATION WITH AMIGA.LIB

A simpler method of creating a task is provided by the amiga.lib Exec support function **CreateTask()**, which can be accessed if your code is linked with amiga.lib.

```
CreateTask(char *name, LONG priority, APTR initialPC, ULONG stacksize)
```

A task created with **CreateTask()** may be removed with the amiga.lib **DeleteTask()** function, or it may simply return when it is finished. **CreateTask()** adds a **MemList** to the **tc_MemEntry** of the task it creates, describing all memory it has allocated for the task, including the task stack and the **Task** structure itself. This memory will be deallocated by Exec when the task is either explicitly removed (**RemTask()** or **DeleteTask()**) or when it exits to Exec's default task removal code (**RemTask()**).

Note that a bug in the **CreateTask()** code caused a failed memory allocation to go unnoticed in V33 and early versions of Release 2 amiga.lib.

If your development language is not linkable with *amiga.lib*, it may provide an equivalent built-in function, or you can create your own based on the *createtask.c* code in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

Depending on the priority of a new task and the priorities of other tasks in the system, the newly added task may begin execution immediately.

> *Sharing Library Pointers* Although in most cases it is possible for a parent task to pass a library base to a child task so the child can use that library, for some libraries, this is not possible. For this reason, the only library base sharable between tasks is Exec's library base.

Here is an example of simple task creation. In this example there is no coordination or communication between the main process and the simple task it has created. A more complex example might use named ports and messages to coordinate the activities and shutdown of two tasks. Because our task is very simple and never calls any system functions which could cause it to be signalled or awakened, we can safely remove the task at any time.

> *Keep This In Mind.* Because the simple task's code is a function in our program, we must stop the subtask before exiting.

```
;/* simpletask.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 simpletask.c
Blink FROM LIB:c.o,simpletask.o TO simpletask LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

simpletask.c - Uses the amiga.lib function CreateTask() to create a simple
subtask.  See the Includes and Autodocs manual for CreateTask() source code
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/tasks.h>
#include <libraries/dos.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>

#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }   /* Disable Lattice CTRL/C handling */
int chkabort(void) {return(0);}
#endif
```

```
#define STACK_SIZE 1000L

/* Task name, pointers for allocated task struct and stack */
struct Task *task = NULL;
char *simpletaskname = "SimpleTask";

ULONG sharedvar;

/* our function prototypes */
void simpletask(void);
void cleanexit(UBYTE *,LONG);

void main(int argc,char **argv)
{
    sharedvar = 0L;

    task = CreateTask(simpletaskname,0,simpletask,STACK_SIZE);
    if(!task)  cleanexit("Can't create task",RETURN_FAIL);

    printf("This program initialized a variable to zero, then started a\n");
    printf("separate task which is incrementing that variable right now,\n");
    printf("while this program waits for you to press RETURN.\n");
    printf("Press RETURN now: ");
    getchar();

    printf("The shared variable now equals %ld\n",sharedvar);

    /* We can simply remove the task we added because our simpletask does not make */
    /* any system calls which could cause it to be awakened or signalled later.    */
    Forbid();
    DeleteTask(task);
    Permit();
    cleanexit("",RETURN_OK);
}

void simpletask()
{
    while(sharedvar < 0x8000000) sharedvar++;
    /* Wait forever because main() is going to RemTask() us */
    Wait(0L);
}

void cleanexit(UBYTE *s, LONG e)
{
    if(*s) printf("%s\n",s);
    exit(e);
}
```

TASK STACK

Every task requires a stack. All task stacks are *user mode* stacks (in the language of the 68000) and are addressed through the A7 CPU register. All normal code execution occurs on this task stack. Special modes of execution (processor traps and system interrupts for example) execute on a single *supervisor mode* stack and do not directly affect task stacks.

Task stacks are normally used to store local variables, subroutine return addresses, and saved register values. Additionally, when a task loses the processor, all of its current registers are preserved on this stack (with the exception of the stack pointer itself, which must be saved in the task structure).

The amount of stack used by a task can vary widely. The theoretical minimum stack size is 72 bytes, which is the number required to save 17 CPU registers and a single return address. Of course, a stack of this size would not give you adequate space to perform any subroutine calls (because the return address occupies stack space). On the other hand, a stack size of 1K would suffice to call most system functions but would not allow much in the way of local variable storage. Processes that call DOS library functions need an additional 1500 bytes of stack.

Because stack-bounds checking is not provided as a service of Exec, it is important to provide enough space for your task stack. Stack overflows are always difficult to debug and may result not only in the erratic failure of your task but also in the mysterious malfunction of other Amiga subsystems. Some compilers provide a stack-checking option.

> *You Can't Always Check The Stack.* Such stack-checking options generally cannot be used if part of your code will be running on the system stack (interrupts, 680x0 exceptions, handlers, servers), or on a different task's stack (libraries, devices, created tasks).

When choosing your stack size, do not cut it too close. Remember that any recursive routines in your code may use varying amounts of stack, and that future versions of system routines may use additional stack variables. By dynamically allocating buffers and arrays, most application programs can be designed to function comfortably within the default process stack size of 4000 bytes.

TASK PRIORITY

A task's priority indicates its importance relative to other tasks. Higher-priority tasks receive the processor before lower-priority tasks do. Task priority is stored as a signed number ranging from -128 to +127. Higher priorities are represented by more positive values; zero is considered the neutral priority. Normally, system tasks execute somewhere in the range of +20 to -20, and most application tasks execute at priority 0.

It is not wise to needlessly raise a task's priority. Sometimes it may be necessary to carefully select a priority so that the task can properly interact with various system tasks. The **SetTaskPri()** Exec function is provided for this purpose.

Task Termination

Task termination may occur as the result of a number of situations:

- A program returning from its **initialPC** routine and dropping into its **finalPC** routine or the system default finalizer.
- A task trap that is too serious for a recovery action. This includes traps like processor bus error, odd address access errors, etc.
- A trap that is not handled by the task. For example, the task might be terminated if your code happened to encounter a processor TRAP instruction and you did not provide a trap handling routine.
- An explicit call to Exec **RemTask()** or amiga.lib **DeleteTask()**.

Task termination involves the deallocation of system resources and the removal of the task structure from Exec. The most important part of task termination is the deallocation of system resources. A task must return all memory that it allocated for its private use, it must terminate any outstanding I/O commands, and it must close access to any system libraries or devices that it has opened.

It is wise to adopt a strategy for task clean-up responsibility. You should decide whether resource allocation and deallocation is the duty of the creator task or the newly created task. Often it is easier and safer for the creator to handle the resource allocation and deallocation on behalf of its offspring. In such cases, before removing the child task, you must make sure it is in a safe state such as **Wait**(0L) and not still using a resources or waiting for an event or signal that might still occur.

NOTE: Certain resources, such as signals and created ports, must be allocated and deallocated by the same task that will wait on them. Also note that if your subtask code is part of your loaded program, you must not allow your program to exit before its subtasks have cleaned up their allocations, and have been either deleted or placed in a safe state such as **Wait**(0L).

Task Exclusion

From time to time the advanced system program may find it necessary to access global system data structures. Because these structures are shared by the system and by other tasks that execute asynchronously to your task, a task must prevent other tasks from using these structures while it is reading from or writing to them. This can be accomplished by preventing the operating system from switching tasks by *forbidding* or *disabling*. A section of code that requires the use of either of these mechanisms to lock out access by others is termed a *critical section*. Use of these methods is discouraged. For arbitrating access to data between your tasks, *semaphores* are a superior solution. (See the ''Exec Semaphores'' chapter)

FORBIDDING TASK SWITCHING

Forbidding is used when a task is accessing shared structures that might also be accessed at the same time from another task. It effectively eliminates the possibility of simultaneous access by imposing *nonpreemptive* task scheduling. This has the net effect of disabling multitasking for as long as your task remains in its running state. While forbidden, your task will continue running until it performs a call to **Wait**() or exits from the forbidden state. Interrupts will occur normally, but no new tasks will be dispatched, *regardless of their priorities.*

When a task running in the forbidden state calls the **Wait**() function, directly or indirectly, it implies a temporary exit from its forbidden state. Since almost all stdio, device I/O, and file I/O functions must **Wait**() for I/O completion, performing such calls will cause your task to **Wait**(), temporarily breaking the forbid. While the task is waiting, the system will perform normally. When the task receives one of the signals it is waiting for, it will again reenter the forbidden state. To become forbidden, a task calls the **Forbid**() function. To escape, the **Permit**() function is used. The use of these functions may be nested with the expected affects; you will not exit the forbidden mode until you call the outermost **Permit**().

As an example, the Exec task list should only be accessed when in a **Forbid**() state. Accessing the list without forbidding could lead to incorrect results or it could crash the entire system. To access the task list also requires the program to disable interrupts which is discussed in the next section.

DISABLING TASKS

Disabling is similar to forbidding, but it also prevents interrupts from occurring during a critical section. Disabling is required when a task accesses structures that are shared by interrupt code. It eliminates the possibility of an interrupt accessing shared structures by preventing interrupts from occurring. Use of disabling is strongly discouraged.

To disable interrupts you can call the **Disable**() function. To enable interrupts again, use the **Enable**() function. Although assembler DISABLE and ENABLE macros are provided, assembler programmers should use the system functions rather than the macros for upwards compatibility, ease of debugging, and smaller code size.

Like forbidden sections, disabled sections can be nested. To restore normal interrupt processing, an **Enable()** call must be made for every **Disable()**. Also like forbidden sections, any direct or indirect call to the **Wait()** function will enable interrupts until the task regains the processor.

> *WARNING:* It is important to realize that there is a danger in using disabled sections. Because the software on the Amiga depends heavily on its interrupts occurring in nearly real time, you cannot disable for more than a very brief instant. Disabling interrupts for more than 250 microseconds can interfere with the normal operation of vital system functions, especially serial I/O.

> *WARNING:* Masking interrupts by changing the 68000 processor interrupt priority levels with the `MOVE SR` instruction can also be dangerous and is very strongly discouraged. The disable- and enable-related functions control interrupts through the 4703 custom chip and *not* through the 68000 priority level. In addition, the processor priority level can be altered only from supervisor mode (which means this process is much less efficient).

It is never necessary to both **Disable()** and **Forbid()**. Because disabling prevents interrupts, it also prevents preemptive task scheduling. When disable is used within an interrupt, it will have the effect of locking out all higher level interrupts (lower level interrupts are automatically disabled by the CPU). Many Exec lists can only be accessed while disabled. Suppose you want to print the names of all system tasks. You would need to access both the **TaskReady** and **TaskWait** lists from within a single disabled section. In addition, you must avoid calling system functions that would break a disable by an indirect call to **Wait()** (**printf()** for example). In this example, the names are gathered into a list while task switching is disabled. Then task switching is enabled and the names are printed.

```
;/* tasklist.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 tasklist.c
Blink FROM LIB:c.o,tasklist.o TO tasklist LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

tasklist.c - Snapshots and prints the ExecBase task list
*/
#include <exec/types.h>
#include <exec/lists.h>
#include <exec/nodes.h>
#include <exec/memory.h>
#include <exec/execbase.h>

#include <clib/alib_protos.h>
#include <clib/exec_protos.h>

#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }   /* disable SAS/C CTRL-C handing */
int chkabort(void) {return(0); }
#endif

static UBYTE *VersTag = "$VER: tasklist 37.2 (31.3.92)";
extern struct ExecBase *SysBase;

/* Use extended structure to hold task information */
struct TaskNode {
    struct Node tn_Node;
    ULONG tn_TaskAddress;
    ULONG tn_SigAlloc;
    ULONG tn_SigWait;
    UBYTE tn_Name[32];
};
```

```
void main(int argc, char **argv)
{
    struct List *ourtasklist;
    struct List *exectasklist;
    struct Task *task;
    struct TaskNode *node, *tnode, *rnode = NULL;
    struct Node *execnode;

    /* Allocate memory for our list */
    if (ourtasklist = AllocMem(sizeof(struct List), MEMF_CLEAR)) {
        /* Initialize list structure (ala NewList()) */
        ourtasklist->lh_Head = (struct Node *)&ourtasklist->lh_Tail;
        ourtasklist->lh_Tail = 0;
        ourtasklist->lh_TailPred = (struct Node *)&ourtasklist->lh_Head;

        /* Make sure tasks won't switch lists or go away */
        Disable();

        /* Snapshot task WAIT list */
        exectasklist = &(SysBase->TaskWait);
        for (execnode = exectasklist->lh_Head;
                 execnode->ln_Succ; execnode = execnode->ln_Succ)
        {
            if (tnode = AllocMem(sizeof(struct TaskNode), MEMF_CLEAR))
            {
                /* Save task information we want to print */
                strncpy(tnode->tn_Name, execnode->ln_Name, 32);
                tnode->tn_Node.ln_Pri = execnode->ln_Pri;
                tnode->tn_TaskAddress = (ULONG)execnode;
                tnode->tn_SigAlloc = ((struct Task *)execnode)->tc_SigAlloc;
                tnode->tn_SigWait = ((struct Task*)execnode)->tc_SigWait;
                AddTail(ourtasklist, (struct Node *)tnode);
            }
            else break;
        }

        /* Snapshot task READY list */
        exectasklist = &(SysBase->TaskReady);
        for (execnode = exectasklist->lh_Head;
                 execnode->ln_Succ; execnode = execnode->ln_Succ)
        {
            if (tnode = AllocMem(sizeof(struct TaskNode), MEMF_CLEAR))
            {
                /* Save task information we want to print */
                strncpy(tnode->tn_Name, execnode->ln_Name, 32);
                tnode->tn_Node.ln_Pri = execnode->ln_Pri;
                tnode->tn_TaskAddress = (ULONG)execnode;
                tnode->tn_SigAlloc = ((struct Task *)execnode)->tc_SigAlloc;
                tnode->tn_SigWait = ((struct Task*)execnode)->tc_SigWait;
                AddTail(ourtasklist, (struct Node *)tnode);
                if(!rnode)  rnode = tnode;  /* first READY task */
            }
            else
                break;
        }

        /* Re-enable interrupts and taskswitching */
        Enable();

        /* Print now (printing above would have defeated a Forbid or Disable) */
        printf("Pri Address     SigAlloc    SigWait    Taskname\n");

        node = (struct TaskNode *)(ourtasklist->lh_Head);
        printf("\nWAITING:\n");
        while (tnode = (struct TaskNode *)node->tn_Node.ln_Succ)
        {
            if(tnode == rnode)
                printf("\nREADY:\n");  /* we set rnode above */
            printf("%02d  0x%08lx  0x%08lx  0x%08lx %s\n",
                    node->tn_Node.ln_Pri, node->tn_TaskAddress, node->tn_SigAlloc,
                    node->tn_SigWait, node->tn_Name);

            /* Free the memory, no need to remove the node, referenced once only */
            FreeMem(node,sizeof(struct TaskNode));
            node = tnode;
        }
```

```
        FreeMem(ourtasklist, sizeof(struct List));

        /* Say who we are */
        printf("\nTHIS TASK:\n");
        task = FindTask(NULL);
        printf("%02d  0x%08lx  0x%08lx  0x%08lx %s\n",
                task->tc_Node.ln_Pri, task, task->tc_SigAlloc,
                task->tc_SigWait, task->tc_Node.ln_Name);

    }
}
```

TASK SEMAPHORES

Semaphores can be used for the purposes of mutual exclusion. With this method of locking, all tasks agree on a locking convention before accessing shared data structures. Tasks that do not require access are not affected and will run normally, so this type of exclusion is considered preferable to forbidding and disabling. This form of exclusion is explained in more detail in the ''Exec Semaphores'' chapter.

Task Exceptions

Exec can provide a task with its own task-local ''interrupt'' called an *exception*. When some exceptional event occurs, an Exec exception occurs which stops a particular task from executing its normal code and forces it to execute a special, task-specific exception handling routine.

If you are familiar with the 680x0, you may be used to using the term ''exceptions'' in a different way. The 680x0 has its own form of exception that has nothing to do with an Exec exception. These are discussed in more detail in the ''Task Traps'' section of this chapter. Do not confuse Exec exceptions with 680x0 exceptions.

To set up an exception routine for a task requires setting values in the task's control structure (the **Task** structure). The **tc_ExceptCode** field should point to the task's exception handling routine. If this field is zero, Exec will ignore all exceptions. The **tc_ExceptData** field should point to any data the exception routine needs.

Exec exceptions work using signals. When a specific signal or signals occur, Exec will stop a task and execute its exception routine. Use the Exec function **SetExcept()** to tell Exec which of the task's signals should trigger the exception.

When an exception occurs, Exec stops executing the tasks normal code and jumps immediately into the exception routine, no matter what the task was doing. The exception routine operates in the same context the task's normal code; it operates in the CPU's user mode and uses the task's stack.

Before entering the exception routine, Exec pushes the normal task code's context onto the stack. This includes the PC, SR, D0-D7, and A0-A6 registers. Exec then puts certain parameters in the processor registers for the exception routine to use. D0 contains a signal mask indicating which signal bit or bits caused the exception. Exec disables these signals when the task enters its exception routine. If more than one signal bit is set (i.e. if two signals occurred simultaneously), it is up to the exception routine to decide in what order to process the two different signals. A1 points to the related exception data (from **tc_ExceptData**), and A6 contains the Exec library base. You can think of an exception as a subtask outside of your normal task. Because task exception code executes in *user* mode, however, the task stack must be large enough to supply the extra space consumed during an exception.

While processing a given exception, Exec prevents that exception from occurring recursively. At exit from your exception-processing code, you should make sure D0 contains the signal mask the exception routine received in D0 because Exec looks here to see which signals it should reactivate. When the task executes the RTS instruction at the end of the exception routine, the system restores the previous contents of all of the task registers and resumes the task at the point where it was interrupted by the exception signal.

> *Exceptions Are Tricky.* Exceptions are difficult to use safely. An exception can interrupt a task that is executing a critical section of code within a system function, or one that has locked a system resource such as the disk or blitter (note that even simple text output uses the blitter.) This possibility makes it dangerous to use most system functions within an exception unless you are sure that your interrupted task was performing only local, non-critical operations.

Task Traps

Task *traps* are synchronous exceptions to the normal flow of program control. They are always generated as a direct result of an operation performed by your program's code. Whether they are accidental or purposely generated, they will result in your program being forced into a special condition in which it must immediately handle the trap. Address error, privilege violation, zero divide, and trap instructions all result in task traps. They may be generated directly by the 68000 processor (Motorola calls them "exceptions") or simulated by software.

A task that incurs a trap has no choice but to respond immediately. The task must have a module of code to handle the trap. Your task may be aborted if a trap occurs and no means of handling it has been provided. Default trap handling code (**tc_TrapCode**) is provided by the OS. You may instead choose to do your own processing of traps. The **tc_TrapCode** field is the address of the handler that you have designed to process the trap. The **tc_TrapData** field is the address of the data area for use by the trap handler.

The system's default trap handling code generally displays a Software Error Requester or Alert containing an exception number and the program counter or task address. Processor exceptions generally have numbers in the range hex 00 to 2F. The 68000 processor exceptions of particular interest are as follows.

Table 21-1: Traps (68000 Exception Vector Numbers)

2	Bus error	access of nonexistent memory
3	Address error	long/word access of odd address (68000)
4	Illegal instruction	illegal opcode (other than Axxx or Fxxx)
5	Zero divide	processor division by zero
6	CHK instruction	register bounds error trap by CHK
7	TRAPV instruction	overflow error trap by TRAPV
8	Privilege violation	user execution of supervisor opcode
9	Trace	status register TRACE bit trap
10	Line 1010 emulator	execution of opcode beginning with $A
11	Line 1111 emulator	execution of opcode beginning with $F
32-47	Trap instructions	TRAP N instruction where N = 0 to 15

A system alert for a processor exception may set the high bit of the longword exception number to indicate an unrecoverable error (for example $80000005 for an unrecoverable processor exception #5). System alerts with more complex numbers are generally Amiga-specific software failures. These are built from the definitions in the <exec/alerts.h> include file.

The actual stack frames generated for these traps are processor-dependent. The 68010, 68020, and 68030 processors will generate a different type of stack frame than the 68000. If you plan on having your program handle its own traps, you should not make assumptions about the format of the supervisor stack frame. Check the flags in the **AttnFlags** field of the **ExecBase** structure for the type of processor in use and process the stack frame accordingly.

TRAP HANDLERS

For compatibility with the 68000, Exec performs trap handling in supervisor mode. This means that all task switching is disabled during trap handling. At entry to the task's trap handler, the system stack contains a processor-dependent trap frame as defined in the 68000/10/20/30 manuals. A longword exception number is added to this frame. That is, when a handler gains control, the top of stack contains the exception number and the trap frame immediately follows.

To return from trap processing, remove the exception number from the stack (note that this is the supervisor stack, not the user stack) and then perform a return from exception (RTE).

Because trap processing takes place in supervisor mode, with task dispatching disabled, it is strongly urged that you keep trap processing as short as possible or switch back to user mode from within your trap handler. If a trap handler already exists when you add your own trap handler, it is smart to propagate any traps that you do not handle down to the previous handler. This can be done by saving the previous address from **tc_TrapCode** and having your handler pass control to that address if the trap which occurred is not one you wish to handle.

The following example installs a simple trap handler which intercepts processor divide-by-zero traps, and passes on all other traps to the previous default trap code. The example has two code modules which are linked together. The trap handler code is in assembler. The C module installs the handler, demonstrates its effectiveness, then restores the previous **tc_TrapCode**.

```
;/* trap_c.c - Execute me to compile me with SAS C 5.10
LC -b0 -cfistq -v -y -j73 trap_c.c
Blink FROM LIB:c.o,trap_c.o,trap_a.o TO trap LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

trap_c.c - C module of sample integer divide-by-zero trap
*/
#include <exec/types.h>
#include <exec/tasks.h>
#include <clib/exec_protos.h>
#include <stdlib.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) {return(0); }
#endif

extern ULONG trapa();           /* assembler trap code in trap_a.asm */

APTR oldTrapCode;
ULONG countdiv0;

void main(int argc, char **argv)
{
    struct Task *thistask;
    ULONG k,j;

    thistask = FindTask(NULL);

    /* Save our task's current trap code pointer */
    oldTrapCode = thistask->tc_TrapCode;
```

```
    /* Point task to our assembler trap handler code.  Ours will just count */
    /* divide-by-zero traps, and pass other traps on to the normal TrapCode */
    thistask->tc_TrapCode = (APTR)trapa;

    countdiv0 = 0L;

    for(k=0; k<4; k++)            /* Let's divide by zero a few times */
       {
       printf("dividing %ld by zero... ",k);
       j = k/0L;
       printf("did it\n");
       }
    printf("\nDivide by zero happened %ld times\n",countdiv0);

    thistask->tc_TrapCode = oldTrapCode;     /* Restore old trap code */
}
```

```
* trap_a.asm - Example trap handling code (leaves D0 intact).  Entered
* in supervisor mode with the following on the supervisor stack:
*    0(sp).l = trap#
*    4(sp) Processor dependent exception frame

    INCLUDE "exec/types.i"
    INCLUDE "libraries/dos.i"

        XDEF _trapa
        XREF _countdiv0
        XREF _oldTrapCode

        CODE
_trapa:                                 ; our trap handler entry
        CMPI.L  #5,(SP)                 ; is this a divide by zero ?
        BNE.S   notdiv0                 ; no
        ADD.L   #1,_countdiv0           ; yes, increment our div0 count
endtrap:
        ADDQ    #4,SP                   ; remove exception number from SSP
        RTE                             ; return from exception
notdiv0:
        TST.L   _oldTrapCode            ; is there another trap handler ?
        BEQ.S   endtrap                 ; no, so we'll exit
        MOVE.L  _oldTrapCode,-(SP)      ; yes, go on to old TrapCode
        RTS                             ; jumps to old TrapCode

        END
```

TRAP INSTRUCTIONS

The TRAP instructions in the 68000 generate traps 32-47. Because many independent pieces of system code may desire to use these traps, the **AllocTrap()** and **FreeTrap()** functions are provided. These work in a fashion similar to that used by **AllocSignal()** and **FreeSignal()**, mentioned above.

Allocating a trap is simply a bookkeeping job within a task. It does not affect how the system calls the trap handler; it helps coordinate who owns what traps. Exec does nothing to determine whether or not a task is prepared to handle a particular trap. It simply calls your code. It is up to your program to handle the trap.

To allocate any trap, you can use the following code:

```
if (-1 == (trap = AllocTrap(-1)))
    printf("all trap instructions are in use\n");
```

Or you can select a specific trap using this code:

```
if (-1 == (trap = AllocTrap(3)))
    printf("trap #3 is in use\n");
```

To free a trap, you use the **FreeTrap()** function passing it the trap number to be freed.

Processor and Cache Control

Exec provides a number of to control the processor mode and, if available, the caches. All these functions work independently of the specific M68000 family processor type. This enables you to write code which correctly controls the state of both the MC68000 and the MC68040. Along with processor mode and cache control, functions are provided to obtain information about the condition code register (CCR) and status register (SR). No functions are provided to control a paged memory management unit (PMMU) or floating point unit (FPU).

Table 21-2: Processor and Cache Control Functions

Function	Description
GetCC()	Get processor condition codes.
SetSR()	Get/set processor status register.
SuperState()	Set supervisor mode with user stack.
Supervisor()	Execute a short supervisor mode function.
UserState()	Return to user mode with user stack.
CacheClearE()	Flush CPU instruction and/or data caches (V37).
CacheClearU()	Flush CPU instruction and data caches (V37).
CacheControl()	Global cache control (V37).
CachePostDMA()	Perform actions prior to hardware DMA (V37).
CachePreDMA()	Perform actions after hardware DMA (V37).

SUPERVISOR MODE

While in supervisor mode, you have complete access to all data and registers, including those used for task scheduling and exceptions, and can execute privileged instructions. In application programs, normally only task trap code is directly executed in supervisor mode, to be compatible with the MC68000. For normal applications, it should never be necessary to switch to supervisor mode itself, only indirectly through Exec function calls. Remember that task switching is disabled while in supervisor mode. If it is absolutely needed to execute code in supervisor mode, keep it as brief as possible.

Supervisor mode can only be entered when a 680x0 exception occurs (an interrupt or trap). The **Supervisor()** function allows you to trap an exception to a specified assembly function. In this function your have full access to all registers. No registers are saved when your function is invoked. You are responsible for restoring the system to a sane state when you are done. You must return to user mode with an RTE instruction. You must not return to user mode by executing a privileged instruction which clears the supervisor bit in the status register. Refer to a manual on the M68000 family of CPUs for information about supervisor mode and available privileged instructions per processor type.

The MC68000 has two stacks, the user stack (USP) and supervisor stack (SSP). As of the MC68020 there are two supervisor stacks, the interrupt stack pointer (ISP) and the master stack pointer (MSP). The **SuperState()** function allows you to enter supervisor mode with the USP used as SSP. The function returns the SSP, which will be the MSP, if an MC68020 or greater is used. Returning to user mode is done with the **UserState()** function. This function takes the SSP as argument, which must be saved when **SuperState()** is called. Because of possible problems with stack size, **Supervisor()** is to be preferred over **SuperState()**.

STATUS REGISTER

The processor status register bits can be set or read with the **SetSR()** function. This function operates in supervisor mode, thus both the upper and lower byte of the SR can be read or set. Be very sure you know what you are doing when you use this function to set bits in the SR and above all never try to use this function to enter supervisor mode. Refer to the M68000 Programmers Reference Manual by Motorola Inc. for information about the definition of individual SR bits per processor type.

CONDITION CODE REGISTER

On the MC68000 a copy of the processor condition codes can be obtained with the MOVE SR,<ea> instruction. On MC68010 processors and up however, the instruction `MOVE CCR,<ea>` must be used. Using the specific MC68000 instruction on later processors will cause a 680x0 exception since it is a privileged instruction on those processors. The **GetCC()** function provides a processor independent way of obtaining a copy of the condition codes. For all processors there are 5 bits which can indicate the result of an integer or a system control instruction:

X - extend N - negative Z - zero V - overflow C - carry

The X bit is used for multiprecision calculations. If used, it is copy of the carry bit. The other bits state the result of a processor operation.

CACHE FUNCTIONS

As of the MC68020 all processors have an instruction cache, 256 bytes on the MC68020 and MC68030 and 4 KBytes on a MC68040. The MC68030 and MC68040 have data caches as well, 256 bytes and 4 KBytes respectively. All the processors load instructions ahead of the program counter (PC), albeit it that the MC68000 and MC68010 only prefetch one and two words respectively. This means the CPU loads instructions ahead of the current program counter. For this reason self-modifying code is strongly discouraged. If your code modifies or decrypts itself just ahead of the program counter, the pre-fetched instructions may not match the modified instructions. If self-modifying code must be used, flushing the cache is the safest way to prevent this.

DMA CACHE FUNCTIONS

The **CachePreDMA()** and **CachePostDMA()** functions allow you to flush the data cache before and after Direct Memory Access. Typically only DMA device drivers benefit from this. These functions take the processor type, possible MMU and cache mode into account. When no cache is available they end up doing nothing. These functions can be replaced with ones suitable for different cache hardware. Refer to the *ROM Kernel Reference Manual: Includes and Autodocs* for implementation specifics.

Since DMA device drivers read and write directly to memory, they are effected by the CopyBack feature of the MC68040 (explained below). Using DMA with CopyBack mode requires a cache flush. If a DMA device needs to read RAM via DMA, it must make sure that the data in the caches has been written to memory first, by calling **CachePreDMA()**. In case of a write to memory, the DMA device should first clear the caches with **CachePreDMA()**, write the data and flush the caches again with **CachePostDMA()**.

THE 68040 AND CPU CACHES

The 68040 is a much more powerful CPU than its predecessors. It has 4K of cache memory for instructions and another 4K cache for data. The reason for these two separate caches is so that the CPU core can access data and CPU instructions *at the same time.*

Although the 68040 provides greater performance it also brings with it greater compatibility problems. Just the fact that the caches are so much larger than Motorola's 68030 CPU can cause problems. However, this is not its biggest obstacle.

The 68040 data cache has a mode that can make the system run *much* faster in most cases. It is called *CopyBack* mode. When a program writes data to memory in this mode, the data goes into the cache but *not* into the physical RAM. That means that if a program or a piece of hardware were to read that RAM without going through the data cache on the 68040, it will read old data. CopyBack mode effects two areas of the Amiga: DMA devices and the CPU's instruction reading.

CopyBack mode effects DMA devices because they read and write data directly to memory. Using DMA with CopyBack mode requires a cache flush. If a DMA device needs to read RAM via DMA, it must first make sure that data in the caches has been written to memory. It can do this by calling the Exec function **CachePreDMA()**. If a DMA device is about to write to memory, it should call **CachePreDMA()** *before* the write, do the DMA write, and then call **CachePostDMA()**, which makes sure that the CPU uses the data just written to memory.

An added advantage of using the **CachePreDMA()** and **CachePostDMA()** functions is that they give the OS the chance to tell the DMA device that the physical addresses and memory sizes are not the same. This will make it possible in the future to add features such as virtual memory. See the Autodocs for more information on these calls.

The other major compatibility problem with the 68040's CopyBack mode is with fetching CPU instructions. CPU instructions have to be loaded into memory so the CPU can copy them into its instruction cache. Normally, instructions that will be executed are written to memory by the CPU (i.e., loading a program from disk). In CopyBack mode, anything the CPU writes to memory, including CPU instructions, doesn't actually go into memory, it goes into the data cache. If instructions are not flushed out of the data cache to RAM, *the 68040 will not be able to find them* when it tries to copy them into the instruction cache for execution. It will instead find and attempt to execute whatever garbage data happened to be left at that location in RAM.

To remedy this, any program that writes instructions to memory must flush the data cache after writing. The V37 Exec function **CacheClearU()** takes care of this. Release 2 of the Amiga OS correctly flushes the caches as needed after it does the **LoadSeg()** of a program (**LoadSeg()** loads Amiga executable programs into memory from disk). Applications need to do the same if they write code to memory. It can do that by calling **CacheClearU()** before the call to **CreateProc()**. In C that would be:

```
extern struct ExecBase *SysBase;

/* If we are in 2.0, call CacheClearU() before CreateProc() */
if (SysBase->LibNode.lib_Version >= 37) CacheClearU();

/* Now do the CreateProc() call... */
proc=CreateProc(... /* whatever your call is like */ ...);
```

For those of you programming in assembly language:

```
*****************************************************************************
* Check to see if we are running in V37 ROM or better.  If so, we want to call
* CacheClearU() to make sure we are safe on future hardware such as the 68040.
* This section of code assumes that a6 points at ExecBase.  a0/a1/d0/d1 are
* trashed in CacheClearU()
*
                cmpi.w  #37,LIB_VERSION(a6)     ; Check if exec is >= V37
                bcs.s   TooOld                  ; If less than V37, too old...
                jsr     _LVOCacheClearU(a6)     ; Clear the cache...
TooOld:                                         ; Exit gracefully.
*****************************************************************************
```

Note that **CreateProc()** is not the only routine where CopyBack mode could be a problem. Any program code copied into memory for execution that is not done via **LoadSeg()** will need to call **CacheClearU()**. Many input device handlers have been known to allocate and copy the handler code into memory and then exit back to the system. These programs also need to have this call in them. The above code will work under older versions of the OS, and will do the correct operations in Release 2 (and beyond).

Function Reference

The following chart gives a brief description of the Exec functions that control tasks. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 21-3: Exec Task, Processor and Cache Control Functions

Exec Task Function	Description
AddTask()	Add a task to the system.
AllocTrap()	Allocate a processor trap vector.
Disable()	Disable interrupt processing.
Enable()	Enable interrupt processing.
FindTask()	Find a specific task.
Forbid()	Forbid task rescheduling.
FreeTrap()	Release a process trap.
Permit()	Permit task rescheduling.
SetTaskPri()	Set the priority of a task.
RemTask()	Remove a task from the system.
CacheClearE()	Flush CPU instruction and/or data caches (V37).
CacheClearU()	Flush CPU instruction and data caches (V37).
CacheControl()	Global cache control (V37).
CachePostDMA()	Perform actions prior to hardware DMA (V37).
CachePreDMA()	Perform actions after hardware DMA (V37).
GetCC()	Get processor condition codes.
SetSR()	Get/set processor status register.
SuperState()	Set supervisor mode with user stack.
Supervisor()	Execute a short supervisor mode function.
UserState()	Return to user mode with user stack.
CreateTask()	Amiga.lib function to setup and add a new task.
DeleteTask()	Amiga.lib function to delete a task created with **CreateTask()**.

Chapter 22
EXEC SIGNALS

Tasks often need to coordinate with other concurrent system activities (like other tasks and interrupts). This coordination is handled by Exec through the synchronized exchange of specific event indicators called *signals*.

This is the primary mechanism responsible for all intertask communication and synchronization on the Amiga. This signal mechanism operates at a low level and is designed for high performance. Signals are used extensively by the Exec message system as a way to indicate the arrival of an inter-task message. The message system is described in more detail in the ''Exec Messages and Ports'' chapter.

> *Not for Beginners.* This chapter concentrates on details about signals that most applications do not need to understand for general Amiga programming. For a general overview of signals, see the ''Introduction to Exec'' chapter of this manual.

The Signal System

The signal system is designed to support independent simultaneous events, so several signals can occur at the same time. Each task has 32 independent signals, 16 of which are pre-allocated for use by the operating system. The signals in use by a particular task are represented as bits in a 32-bit field in its **Task** structure (<exec/tasks.h>). Two other 32-bit fields in the **Task** structure indicate which signals the task is waiting for, and which signals have been received.

Signals are *task relative*. A task can only allocate its own signals, and may only wait on its own signals. In addition, a task may assign its own significance to a particular signal. Signals are not broadcast to all tasks; they are directed only to individual tasks. A signal has meaning to the task that defined it and to those tasks that have been informed of its meaning.

For example, signal bit 12 may indicate a timeout event to one task, but to another task it may indicate a message arrival event. You can never wait on a signal that you did not directly or indirectly allocate yourself, and any other task that wishes to signal you must use a signal that *you* allocated.

SIGNAL ALLOCATION

As mentioned above, a task assigns its own meaning to a particular signal. Because certain system libraries may occasionally require the use of a signal, there is a convention for signal allocation. It is unwise ever to make assumptions about which signals are actually in use.

Before a signal can be used, it must be allocated with the **AllocSignal()** function. When a signal is no longer needed, it should be freed for reuse with **FreeSignal()**.

```
BYTE AllocSignal( LONG signalNum );
VOID FreeSignal( LONG signalNum );
```

AllocSignal() marks a signal as being in use and prevents the accidental use of the same signal for more than one event. You may ask for either a specific signal number, or more commonly, you would pass -1 to request the next available signal. The state of the newly allocated signal is cleared (ready for use). Generally it is best to let the system assign you the next free signal. Of the 32 available signals, the lower 16 are reserved for system use. This leaves the upper 16 signals free for application programs to allocate. Other subsystems that you may call depend on **AllocSignal()**.

The following C example asks for the next free signal to be allocated for its use:

```
if (-1 == (signal = AllocSignal(-1)))
    printf("no signal bits available\n");
else
    {
    printf("allocated signal number %ld\n", signal);
    /* Other code could go here */
    FreeSignal(signal)
    }
```

The value returned by **AllocSignal()** is a signal bit number. This value cannot be used directly in calls to signal-related functions without first being converted to a mask:

```
mask = 1L << signal;
```

It is important to realize that signal bit allocation is relevant *only* to the running task. You *cannot* allocate a signal from another task. Note that functions which create a signal **MsgPort** will allocate a signal from the task that calls the function. Such functions include **OpenWindow()**, **CreatePort()**, and **CreateMsgPort()**. For this reason, only the creating task may **Wait()** (directly or indirectly) on the **MsgPort**'s signal. Functions which call **Wait()** include **DoIO()**, **WaitIO()** and **WaitPort()**.

WAITING FOR A SIGNAL

Signals are most often used to wake up a task upon the occurrence of some external event. Applications call the Exec **Wait()** function, directly or indirectly, in order to enter a wait state until some external event triggers a signal which awakens the task.

Though signals are usually not used to interrupt an executing task, they can be used this way. Task *exceptions*, described in the ''Exec Interrupts'' chapter, allow signals to act as a task-local interrupt.

The **Wait()** function specifies the set of signals that will wake up the task and then puts the task to sleep (into the waiting state).

```
ULONG Wait( ULONG signalSet );
```

Any one signal or any combination of signals from this set are sufficient to awaken the task. **Wait()** returns a mask indicating which signals satisfied the **Wait()** call. Note that when signals are used in conjunction with a message port, a set signal bit does not necessarily mean that there is a message at the message port. See the ''Exec Messages and Ports'' chapter for details about proper handling of messages.

Because tasks (and interrupts) normally execute asynchronously, it is often possible to receive a particular signal before a task actually **Wait()**s for it. In such cases the **Wait()** will be immediately satisfied, and the task will not be put to sleep.

The **Wait()** function implicitly clears those signal bits that satisfied the wait condition. This effectively resets those signals for reuse. However, keep in mind that a task might get more signals while it is still processing the previous signal. If the same signal is received multiple times and the signal bit is not cleared between them, some signals will go unnoticed.

Be aware that using **Wait()** will break a **Forbid()** or **Disable()** state. **Wait()** cannot be used in supervisor mode or within interrupts.

A task may **Wait()** for a combination of signal bits and will wake up when any of the signals occur. **Wait()** returns a signal mask specifying which signal or signals were received. Usually the program must check the returned mask for each signal it was waiting on and take the appropriate action for each that occurred. The order in which these bits are checked is often important.

Here is a hypothetical example of a process that is using the console and timer devices, and is waiting for a message from either device and a possible break character issued by the user:

```
consoleSignal = 1L << ConsolePort->mp_SigBit;
timerSignal   = 1L << TimerPort->mp_SigBit;
userSignal    = SIGBREAKF_CTRL_C;              /* Defined in <dos/dos.h> */

signals = Wait(consoleSignal | timerSignal | userSignal);

if (signals & consoleSignal)
    printf("new character\n");

if (signals & timeOutSignal)
    printf("timeout\n");

if (signals & userSignal)
    printf("User Ctrl-C Abort\n");
```

This code will put the task to sleep waiting for a new character, or the expiration of a time period, or a Ctrl-C break character issued by the user. Notice that this code checks for an incoming character signal before checking for a timeout. Although a program can check for the occurrence of a particular event by checking whether its signal has occurred, this may lead to busy wait polling. Such polling is wasteful of the processor and is usually harmful to the proper function of the Amiga system. However, if a program needs to do constant processing and also check signals (a compiler for example) **SetSignal**(0,0) can be used to get a copy of your task's current signals.

```
ULONG SetSignal( ULONG newSignals, ULONG signalSet );
```

SetSignal() can also be used to set or clear the state of the signals. Implementing this can be dangerous and should generally not be done. The following fragment illustrates a possible use of SetSignal().

```
signals = SetSignal(0,0);            /* Get current state of signals */

if (signals & SIGBREAKF_CTRL_C)      /* Check for Ctrl-C.           */
    {
    printf("Break\n");              /* Ctrl-C signal has been set. */
    SetSignal(0, SIGBREAKF_CTRL_C)   /* Clear Ctrl-C signal.        */
    }
```

GENERATING A SIGNAL

Signals may be generated from both tasks and system interrupts with the **Signal()** function.

```
VOID Signal( struct Task *task, ULONG signalSet );
```

For example **Signal**(tc,mask) would signal the task with the specified mask signals. More than one signal can be specified in the mask. The following example code illustrates **Wait()** and **Signal()**.

```
;/* signals.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 signals.c
Blink FROM LIB:c.o,signals.o TO signals LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/alib_protos.h>
#include "stdio.h"

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

static UBYTE *VersTag = "$VER: signals 37.1 (28.3.91)";

void subtaskcode(void);    /* prototype for our subtask routine */

LONG  mainsignum = -1;
ULONG mainsig, wakeupsigs;
struct Task *maintask = NULL, *subtask = NULL;
UBYTE subtaskname[] = "RKM_signal_subtask";

void main(int argc, char **argv)
{
BOOL Done = FALSE, WaitingForSubtask = TRUE;

/* We must allocate any special signals we want to receive. */
mainsignum = AllocSignal(-1);
if(mainsignum == -1)
    printf("No signals available\n");
else
    {
    mainsig = 1L << mainsignum;     /* subtask can access this global */
    maintask = FindTask(NULL);      /* subtask can access this global */

    printf("We alloc a signal, create a task, wait for signals\n");
    subtask = CreateTask(subtaskname, 0L, subtaskcode, 2000);
    if(!subtask)
        printf("Can't create subtask\n");
    else
        {
        printf("After subtask signals, press CTRL-C or CTRL-D to exit\n");
```

```
        while((!Done) || (WaitingForSubtask))
            {
            /* Wait on the combined mask for all of the signals we are
             * interested in.  All processes have the CTRL_C thru CTRL_F
             * signals.  We're also Waiting on the mainsig we allocated
             * for our subtask to signal us with.  We could also Wait on
             * the signals of any ports/windows our main task created ... */

            wakeupsigs = Wait(mainsig | SIGBREAKF_CTRL_C | SIGBREAKF_CTRL_D);

            /* Deal with all signals that woke us up - may be more than one */
            if(wakeupsigs & mainsig)
                {
                printf("Signalled by subtask\n");
                WaitingForSubtask = FALSE;   /* OK to kill subtask now */
                }
            if(wakeupsigs & SIGBREAKF_CTRL_C)
                {
                printf("Got CTRL-C signal\n");
                Done = TRUE;
                }
            if(wakeupsigs & SIGBREAKF_CTRL_D)
                {
                printf("Got CTRL-D signal\n");
                Done = TRUE;
                }
            }
        Forbid();
        DeleteTask(subtask);
        Permit();
        }
    FreeSignal(mainsignum);
    }
}

void subtaskcode(void)
    {
    Signal(maintask,mainsig);
    Wait(0L);    /* safe state in which this subtask can be deleted */
    }
```

Function Reference

The following chart gives a brief description of the Exec functions that control task signalling. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 22-1: Exec Signal Functions

Exec Signal Function	Description
AllocSignal()	Allocate a signal bit.
FreeSignal()	Free a signal bit allocated with **AllocSignal()**.
SetSignal()	Query or set the state of the signals for the current task.
Signal()	Signal a task by setting signal bits in its **Task** structure.
Wait()	Wait for one or more signals from other tasks or interrupts.

Chapter 23
EXEC LISTS AND QUEUES

The Amiga system software operates in a dynamic environment of data structures. An early design goal of Exec was to keep the system flexible and open-ended by eliminating artificial boundaries on the number of system structures used. Rather than using static system tables, Exec uses dynamically created structures that are added and removed as needed. These structures can be put in an unordered *list*, or in an ordered *list* known as a *queue*. A list can be empty, but never full. This concept is central to the design of Exec. Understanding lists and queues is important to understanding not only Exec itself, but also the mechanism behind the Amiga's message and port based interprocess communication.

Exec uses lists to maintain its internal database of system structures. Tasks, interrupts, libraries, devices, messages, I/O requests, and all other Exec data structures are supported and serviced through the consistent application of Exec's list mechanism. Lists have a common data structure, and a common set of functions is used for manipulating them. Because all of these structures are treated in a similar manner, only a small number of list handling functions need be supported by Exec.

List Structure

A list is composed of a *header* and a doubly-linked chain of elements called *nodes*. The header contains memory pointers to the first and last nodes of the linked chain. The address of the header is used as the handle to the entire list. To manipulate a list, you must provide the address of its header.

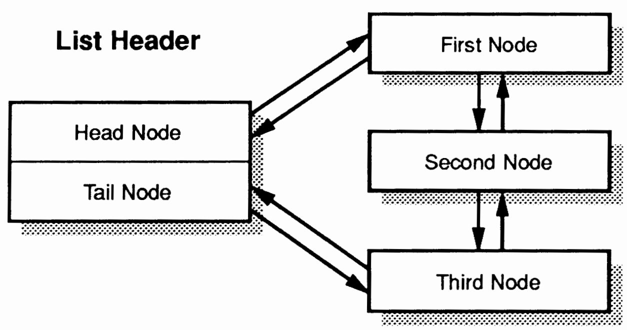


Figure 23-1: Simplified Overview of an Exec List

Nodes may be scattered anywhere in memory. Each node contains two pointers; a successor and a predecessor. As illustrated above, a list header contains two placeholder nodes that contain no data. In an empty list, the head and tail nodes point to each other.

NODE STRUCTURE DEFINITION

A **Node** structure is divided into three parts: linkage, information, and content. The linkage part contains memory pointers to the node's successor and predecessor nodes. The information part contains the node type, the priority, and a name pointer. The content part stores the actual data structure of interest. For nodes that require linkage only, a small **MinNode** structure is used.

```
struct MinNode
{
    struct MinNode *mln_Succ;
    struct MinNode *mln_Pred;
};
```

ln_Succ
points to the next node in the list (successor).

ln_Pred
points to the previous node in the list (predecessor).

When a type, priority, or name is required, a full-featured **Node** structure is used.

```
struct Node
{
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE        ln_Type;
    BYTE         ln_Pri;
    char        *ln_Name;
};
```

ln_Type
defines the type of the node (see *<exec/nodes.h>* for a list).

ln_Pri
specifies the priority of the node (+127 (highest) to -128 (lowest)).

ln_Name
points to a printable name for the node (a NULL-terminated string).

The **Node** and **MinNode** structures are often incorporated into larger structures, so groups of the larger structures can easily be linked together. For example, the Exec **Interrupt** structure is defined as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR        is_Data;
    VOID        (*is_Code)();
};
```

Here the **is_Data** and **is_Code** fields represent the useful content of the node. Because the **Interrupt** structure begins with a **Node** structure, it may be passed to any of the Exec **List** manipulation functions.

NODE INITIALIZATION

Before linking a node into a list, certain fields may need initialization. Initialization consists of setting the **ln_Type**, **ln_Pri**, and **ln_Name** fields to their appropriate values (A **MinNode** structure does not have these fields). The successor and predecessor fields do not require initialization.

The **ln_Type** field contains the data type of the node. This indicates to Exec (and other subsystems) the type, and hence the structure, of the content portion of the node (the extra data after the **Node** structure). The standard system types are defined in the *<exec/nodes.h>* include file. Some examples of standard system types are NT_TASK, NT_INTERRUPT, NT_DEVICE, and NT_MSGPORT.

The **ln_Pri** field uses a signed numerical value ranging from +127 to -128 to indicate the priority of the node. Higher-priority nodes have greater values; for example, 127 is the highest priority, zero is nominal priority, and -128 is the lowest priority. Some Exec lists are kept sorted by priority order. In such lists, the highest-priority node is at the head of the list, and the lowest-priority node is at the tail of the list. Most Exec node types do not use a priority. In such cases, initialize the priority field to zero.

The **ln_Name** field is a pointer to a NULL-terminated string of characters. Node names are used to find and identify list-bound objects (like public message ports and libraries), and to bind symbolic names to actual nodes. Names are also useful for debugging purposes, so it is a good idea to provide every node with a name. Take care to provide a valid name pointer; Exec does not copy name strings.

This fragment initializes a **Node** called **myInt**, an instance of the **Interrupt** data structure introduced above.

```
    struct Interrupt interrupt;

    interrupt.is_Node.ln_Type = NT_INTERRUPT;
    interrupt.is_Node.ln_Pri  = -10;
    interrupt.is_Node.ln_Name = "sample.interrupt";
```

LIST HEADER STRUCTURE DEFINITION

As mentioned earlier, a list header maintains memory pointers to the first and last nodes of the linked chain of nodes. It also serves as a handle for referencing the entire list. The minimum list header (''mlh_'') and the full-featured list header (''lh_'') are generally interchangeable.

The structure **MinList** defines a minimum list header.

```
struct MinList
{
    struct MinNode *mlh_Head;
    struct MinNode *mlh_Tail;
    struct MinNode *mlh_TailPred;
};
```

mlh_Head
: points to the first node in the list.

mlh_Tail
: is always NULL.

mlh_TailPred
: points to the last node in the list.

In a few limited cases a full-featured **List** structure will be required:

```
struct List
{
    struct Node *lh_Head;
    struct Node *lh_Tail;
    struct Node *lh_TailPred;
    UBYTE        lh_Type;
    UBYTE        lh_Pad;
};
```

lh_Type
defines the type of nodes within the list (see *<exec/nodes.h>*).

lh_pad
is a structure alignment byte.

One subtlety here must be explained further. The list header is constructed in an efficient, but confusing manner. Think of the header as a structure containing the head and tail nodes for the list. The head and tail nodes are placeholders, and never carry data. The head and tail portions of the header actually overlap in memory. **lh_Head** and **lh_Tail** form the head node; **lh_Tail** and **lh_TailPred** form the tail node. This makes it easy to find the start or end of the list, and eliminates any special cases for insertion or removal.

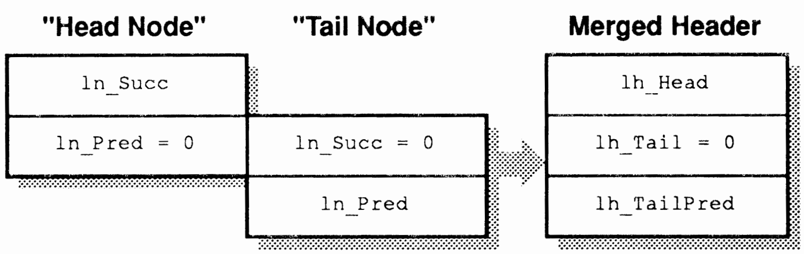


Figure 23-2: List Header Overlap

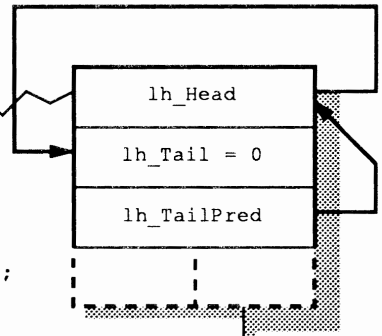
The **lh_Head** and **lh_Tail** fields of the list header act like the **ln_Succ** and **lh_Pred** fields of a node. The **lh_Tail** field is set permanently to NULL, indicating that the head node is indeed the first on the list — that is, it has no predecessors. See the figure below.

Likewise, the **lh_Tail** and **lh_TailPred** fields of the list header act like the **ln_Succ** and **lh_Pred** fields of a node. Here the NULL **lh_Tail** indicates that the tail node is indeed the last on the list — that is, it has no successors. See the figure below.

HEADER INITIALIZATION

List headers must be properly initialized before use. It is not adequate to initialize the entire header to zero. The head and tail entries must have specific values. The header must be initialized as follows:

1. Set the **lh_Head** field to the address of **lh_Tail**.
2. Clear the **lh_Tail** field.
3. Set the **lh_TailPred** field to the address of **lh_Head**.
4. Set **lh_Type** to the same data type as the nodes to be kept the list. (Unless you are using a **MinList**).




```
/* C example - equivalent to NewList() */
struct List list;

list.lh_Head          = (struct Node *) &list.lh_Tail;
list.lh_Tail          = 0;
list.lh_TailPred      = (struct Node*) &list.lh_Head;
/* Now set lh_Type, if needed */

;Assembly example - equivalent to NEWLIST
       MOVE.L  A0,LH_HEAD(A0) ;A0 points to the list header
       ADDQ.L  #4,LH_HEAD(A0) ;Bump LH_HEAD(A0) to address of LH_TAIL
       CLR.L   LH_TAIL(A0)
       MOVE.L  A0,LH_TAILPRED(A0)
;Now set LH_TYPE, if needed.
```

Figure 23-3: Initializing a List Header Structure

The sequence of assembly instructions in the figure above is equivalent to the macro **NEWLIST**, contained in the include file *<exec/lists.i>*. Since the **MinList** structure is the same as the **List** structure except for the type and pad fields, this sequence of assembly language code will work for both structures. The sequence performs its function without destroying the pointer to the list header in A0 (which is why **ADDQ.L** is used). This function may also be accessed from C as a call to **NewList(header)**, where **header** is the address of a list header.

List Functions

Exec provides a number of symmetric functions for handling lists. There are functions for inserting and removing nodes, for adding and removing head and tail nodes, for inserting nodes in a priority order, and for searching for nodes by name. The prototypes for Exec list handling functions are as follows.

Exec Functions

```
VOID AddHead( struct List *list, struct Node *node );
VOID AddTail( struct List *list, struct Node *node );
VOID Enqueue( struct List *list, struct Node *node );
struct Node *FindName( struct List *list, UBYTE *name );
VOID Insert( struct List *list, struct Node *node, struct Node *pred );
VOID Remove( struct Node *node );
struct Node *RemHead( struct List *list );
struct Node *RemTail( struct List *list );
```

Exec Support Functions in *amiga.lib*

```
VOID NewList( struct List *list );
```

In this discussion of the Exec list handling functions, **header** represents a pointer to **List** header, and **node** represents pointer to a **Node**.

INSERTION AND REMOVAL

The **Insert()** function is used for inserting a new node into any position in a list. It always inserts the node following a specified node that is already part of the list. For example, **Insert**(header,node,pred) inserts the node **node** after the node **pred** in the specified list. If the **pred** node points to the list header or is NULL, the new node will be inserted at the head of the list. Similarly, if the **pred** node points to the **lh_Tail** of the list, the new node will be inserted at the tail of the list. However, both of these actions can be better accomplished with the functions mentioned in the ''Special Case Insertion'' section below.

The **Remove()** function is used to remove a specified node from a list. For example, **Remove**(node) will remove the specified node from whatever list it is in. *To be removed, a node must actually be in a list.* If you attempt to remove a node that is not in a list, you will cause serious system problems.

SPECIAL CASE INSERTION

Although the **Insert()** function allows new nodes to be inserted at the head and the tail of a list, the **AddHead()** and **AddTail()** functions will do so with higher efficiency. Adding to the head or tail of a list is common practice in first-in-first-out (FIFO) or last-in-first-out (LIFO or stack) operations. For example, **AddHead**(header,node) would insert the node at the head of the specified list.

SPECIAL CASE REMOVAL

The two functions **RemHead()** and **RemTail()** are used in combination with **AddHead()** and **AddTail()** to create special list ordering. When you combine **AddTail()** and **RemHead()**, you produce a first-in-first-out (FIFO) list. When you combine **AddHead()** and **RemHead()** a last-in-first-out (LIFO or stack) list is produced. **RemTail()** exists for symmetry. Other combinations of these functions can also be used productively.

Both **RemHead()** and **RemTail()** remove a node from the list, and return a pointer to the removed node. If the list is empty, the function return a NULL result.

MINLIST / MINNODE OPERATIONS

All of the above functions and macros will work with long or short format node structures. A **MinNode** structure contains only linkage information. A full **Node** structure contains linkage information, as well as type, priority and name fields. The smaller **MinNode** is used where space and memory alignment issues are important. The larger **Node** is used for queues or lists that require a name tag for each node.

PRIORITIZED INSERTION

The list functions discussed so far do not make use of the priority field in a **Node**. The **Enqueue()** function is equivalent to **Insert()**, except it inserts nodes into a list sorting them according to their priority. It keeps the higher-priority nodes towards the head of the list. All nodes passed to this function must have their priority and name assigned prior to the call. **Enqueue**(header,mynode) inserts **mynode** behind the lowest priority node with a priority greater than or equal to **mynode**'s. For **Enqueue()** to work properly, the list must already be sort according to priority. Because the highest priority node is at the head of the list, the **RemHead()** function will remove the highest-priority node. Likewise, **RemTail()** will remove the lowest-priority node.

FIFO Is Used For The Same Priority. If you add a node that has the same priority as another node in the queue, **Enqueue()** will use FIFO ordering. The new node is inserted following the last node of equal priority.

SEARCHING BY NAME

Because many lists contain nodes with symbolic names attached (via the **ln_Name** field), it is possible to find a node by its name. This naming technique is used throughout Exec for such nodes as tasks, libraries, devices, and resources.

The **FindName()** function searches a list for the first node with a given name. For example, **FindName**(header, "Furrbol") returns a pointer to the first node named ''Furrbol.'' If no such node exists, a NULL is returned. The case of the name characters is significant; ''foo'' is different from ''Foo.''

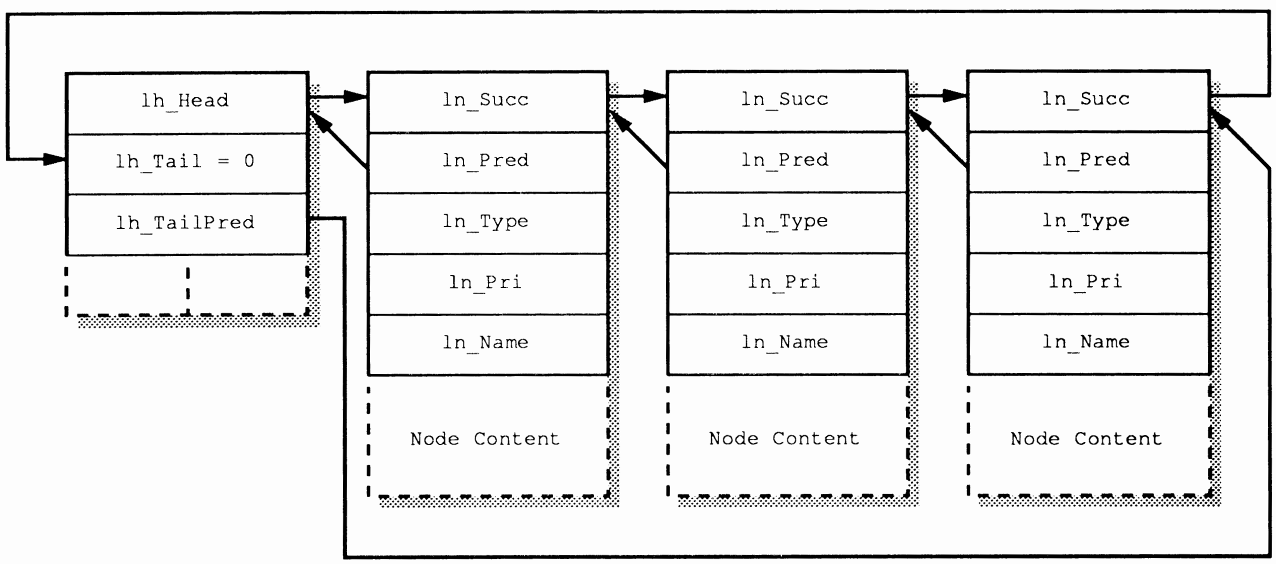


Figure 23-4: Complete Sample List Showing all Interconnections

MORE ON THE USE OF NAMED LISTS

To find multiple occurrences of nodes with identical names, the **FindName()** function is called multiple times. For example, if you want to find all the nodes with the name pointed to by **name**:

```
VOID DisplayName(struct List *list,UBYTE *name)
{
struct Node *node;

    if (node = FindName(list,name))
        while (node)
        {
            printf("Found %s at location %lx\n",node->ln_Name,node);
            node = FindName((struct List *)node,name);
        }
    else printf("No node with name %s found.\n",name);
}
```

Notice that the second search uses the node found by the first search. The **FindName()** function *never* compares the specified name with that of the starting node. It always begins the search with the successor of the starting point.

LIST MACROS FOR ASSEMBLY LANGUAGE PROGRAMMERS

Assembly language programmers may want to optimize their code by using assembly code list macros. Because these macros actually embed the specified list operation into the code, they result in slightly faster operations. The file *<exec/lists.i>* contains the recommended set of macros. For example, the following instructions implement the **REMOVE** macro:

```
MOVE.L  LN_SUCC(A1),A0        ; get successor
MOVE.L  LN_PRED(A1),A1        ; get predecessor
MOVE.L  A0,LN_SUCC(A1)        ; fix up predecessor's succ pointer
MOVE.L  A1,LN_PRED(A0)        ; fix up successor's pred pointer
```

EMPTY LISTS

It is often important to determine if a list is empty. This can be done in many ways, but only two are worth mentioning. If either the **lh_TailPred** field is pointing to the list header or the **ln_Succ** field of the **lh_Head** is NULL, then the list is empty.

In C, for example, these methods would be written as follows:

```
/* You can use this method...   */
if (list->lh_TailPred == (struct Node *)list)
    printf("list is empty\n");

/* Or you can use this method   */
if (NULL == list->lh_Head->ln_Succ)
    printf("list is empty\n");
```

In assembly code, if A0 points to the list header, these methods would be written as follows:

```
; Use this method...
CMP.L   LH_TAILPRED(A0),A0
BEQ     list_is_empty

; Or use this method
MOVE.L  LH_HEAD(A0),A1
TST.L   LN_SUCC(A1)
BEQ     list_is_empty
```

Because **LH_HEAD** and **LN_SUCC** are both zero offsets, the second case may be simplified or optimized by your assembler.

SCANNING A LIST

Occasionally a program may need to scan a list to locate a particular node, find a node that has a field with a particular value, or just print the list. Because lists are linked in both the forward and backward directions, the list can be scanned from either the head or tail.

Here is a code fragment that uses a *for* loop to print the names of all nodes in a list:

```
struct List *list;
struct Node *node;

for (node = list->lh_Head ; node->ln_Succ ; node = node->ln_Succ)
    printf("%lx -> %s\n",node,node->ln_Name);
```

A common mistake is to process the head or tail nodes. Valid data nodes have non-NULL successor and predecessor pointers. The above loop exits when **node->ln_Succ** is NULL. Another common mistake is to free a node from within a loop, then reference the free memory to obtain the next node pointer. An extra temporary pointer solves this second problem.

In assembly code, it is more efficient to use a look-ahead cache pointer when scanning a list. In this example the list is scanned until the first zero-priority node is reached:

```
        MOVE.L  (A1),D1         ; first node
scan:   MOVE.L  D1,A1
        MOVE.L  (A1),D1         ; lookahead to next
        BEQ.S   not_found       ; end of list...
        TST.B   LN_PRI(A1)
        BNE.S   scan
        ...                     ; found one

not_found:
```

Exec List Example

The code below demonstrates the concepts and functions discussed in this chapter by building an application-defined Exec **List**.

```
;/* buildlist.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 buildlist.c
Blink FROM LIB:c.o,buildlist.o TO buildlist LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

buildlist.c - example which uses an application-specific Exec list
*/

#include <exec/types.h>
#include <exec/lists.h>
#include <exec/nodes.h>
#include <exec/memory.h>

#include <clib/alib_protos.h>
#include <clib/exec_protos.h>

#include <string.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)    { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

/* Our function prototypes */
VOID AddName(struct List *, UBYTE *);
VOID FreeNameNodes(struct List *);
VOID DisplayNameList(struct List *);
VOID DisplayName(struct List *, UBYTE *);

struct NameNode {
    struct Node nn_Node;        /* System Node structure */
    UBYTE   nn_Data[62];        /* Node-specific data */
};

#define NAMENODE_ID   100       /* The type of "NameNode" */
```

```
VOID main(VOID)
{
    struct  List    *NameList;    /* Note that a MinList would also work */

    if (!( NameList = AllocMem(sizeof(struct List),MEMF_CLEAR)) )
        printf("Out of memory\n");
    else {
        NewList(NameList);        /* Important: prepare header for use */

        AddName(NameList,"Name7");   AddName(NameList,"Name6");
        AddName(NameList,"Name5");   AddName(NameList,"Name4");
        AddName(NameList,"Name2");   AddName(NameList,"Name0");

        AddName(NameList,"Name7");   AddName(NameList,"Name5");
        AddName(NameList,"Name3");   AddName(NameList,"Name1");

        DisplayName(NameList,"Name5");
        DisplayNameList(NameList);

        FreeNameNodes(NameList);
        FreeMem(NameList,sizeof(struct List));  /* Free list header */
    }
}

/* Allocate a NameNode structure, copy the given name into the structure,
 * then add it the specified list.  This example does not provide an
 * error return for the out of memory condition.
*/
VOID AddName(struct List *list, UBYTE *name)
{
    struct NameNode *namenode;

    if (!( namenode = AllocMem(sizeof(struct NameNode),MEMF_CLEAR) ))
        printf("Out of memory\n");
    else {
        strcpy(namenode->nn_Data,name);
        namenode->nn_Node.ln_Name = namenode->nn_Data;
        namenode->nn_Node.ln_Type = NAMENODE_ID;
        namenode->nn_Node.ln_Pri  = 0;
        AddHead((struct List *)list,(struct Node *)namenode);
    }
}

/*
 * Free the entire list, including the header.  The header is not updated
 * as the list is freed.  This function demonstrates how to avoid
 * referencing freed memory when deallocating nodes.
 */
VOID FreeNameNodes(struct List *list)
{
    struct NameNode *worknode;
    struct NameNode *nextnode;

    worknode = (struct NameNode *)(list->lh_Head); /* First node */
    while (nextnode = (struct NameNode *)(worknode->nn_Node.ln_Succ)) {
        FreeMem(worknode,sizeof(struct NameNode));
        worknode = nextnode;
    }

}

/*
 * Print the names of each node in a list.
 */
VOID DisplayNameList(struct List *list)
{
struct Node *node;

    if (list->lh_TailPred == (struct Node *)list)
        printf("List is empty.\n");
    else {
        for (node = list->lh_Head ; node->ln_Succ ; node = node->ln_Succ)
            printf("%lx -> %s\n",node,node->ln_Name);
    }
}
```

```
/*
 * Print the location of all nodes with a specified name.
 */
VOID DisplayName(struct List *list, UBYTE *name)
{
struct Node *node;

    if (node = FindName(list,name)) {
        while (node) {
            printf("Found a %s at location %lx\n",node->ln_Name,node);
            node = FindName((struct List *)node,name);
        }
    } else printf("No node with name %s found.\n",name);
}
```

IMPORTANT NOTE ABOUT SHARED LISTS

It is possible to run into contention problems with other tasks when manipulating a list that is shared by more than one task. *None* of the standard Exec list functions arbitrates for access to the list. For example, if some other task happens to be modifying a list while your task scans it, an inconsistent view of the list may be formed. This can result in a corrupted system.

Generally it is not permissible to read or write a shared list without first locking out access from other tasks. *All* users of a list must use the same arbitration method. Several arbitration techniques are used on the Amiga. Some lists are protected by a semaphore. The **ObtainSemaphore()** call grants ownership of the list (see the "Exec Semaphores" chapter for more information). Some lists require special arbitration. For example, you must use the Intuition **LockIBase(0)** call before accessing any Intuition lists. Other lists may be accessed only during **Forbid()** or **Disable()** (see the "Exec Tasks" chapter for more information).

The preferred method for arbitrating use of a shared list is through semaphores because a semaphores only holds off other tasks that are trying to access the shared list. Rather than suspending all multitasking. Failure to lock a shared list before use *will* result in unreliable operation.

Note that I/O functions including **printf()** generally call **Wait()** to wait for I/O completion, and this allows other tasks to run. Therefore, it is not safe to print or **Wait()** while traversing a list unless the list is fully controlled by your application, or if the list is otherwise guaranteed not to change during multitasking.

Function Reference

The following charts give a brief description of the Exec list and queue functions and assembler macros. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 23-1: Exec List and Queue Functions

Exec Function	Description
AddHead()	Insert a node at the head of a list.
AddTail()	Append a node to the tail of a list.
Enqueue()	Insert or append a node to a system queue.
FindName()	Find a node with a given name in a system list.
Insert()	Insert a node into a list.
IsListEmpty	Test if list is empty
NewList()	Initialize a list structure for use.
RemHead()	Remove the head node from a list.
Remove()	Remove a node from a list.
RemTail()	Remove the tail node from a list.

Table 23-2: Exec List and Queue Assembler Macros

Exec Function	Description
NEWLIST	Initialize a list header for use.
TSTLIST	Test if list is empty (list address in register). No arbitration needed.
TSTLST2	Test is list is empty (from effective address of list). Arbitration needed.
SUCC	Get next node in a list.
PRED	Get previous node in a list.
IFEMPTY	Branch if list is empty.
IFNOTEMPTY	Branch if list is not empty.
TSTNODE	Get next node, test if at end of list.
NEXTNODE	Get next node, go to exit label if at end.
ADDHEAD	Add node to head of list.
ADDTAIL	Add node to tail of list.
REMOVE	Remove node from a list.
REMHEAD	Remove node from head of list.
REMHEADQ	Remove node from head of list quickly.
REMTAIL	Remove node from tail of list.

Chapter 24
EXEC MESSAGES AND PORTS

For interprocess communication, Exec provides a consistent, high-performance mechanism of messages and ports. This mechanism is used to pass message structures of arbitrary sizes from task to task, interrupt to task, or task to software interrupt. In addition, messages are often used to coordinate operations between cooperating tasks. This chapter describes many of the details of using messages and ports that the casual Amiga programmer won't need. See the "Introduction to Exec" chapter of this manual for a general introduction to using messages and ports.

A *message* data structure has two parts: system linkage and message body. The system linkage is used by Exec to attach a given message to its destination. The message body contains the actual data of interest. The message body is any arbitrary data up to 64K bytes in size. The message body data can include pointers to other data blocks of any size.

Messages are always sent to a predetermined destination *port*. At a port, incoming messages are queued in a first-in-first-out (FIFO) order. There are no system restrictions on the number of ports or the number of messages that may be queued to a port (other than the amount of available system memory).

Messages are always queued by *reference*, i.e., by a pointer to the message. For performance reasons message copying is not performed. In essence, a message between two tasks is a temporary license for the receiving task to use a portion of the memory space of the sending task; that portion being the message itself. This means that if task A sends a message to task B, the message is still part of the task A context. Task A, however, should not access the message until it has been *replied*; that is, until task B has sent the message back, using the **ReplyMsg()** function. This technique of message exchange imposes important restrictions on message access.

Message Ports

Message ports are rendezvous points at which messages are collected. A port may contain any number of outstanding messages from many different originators. When a message arrives at a port, the message is appended to the end of the list of messages for that port, and a prespecified arrival action is invoked. This action may do nothing, or it may cause a predefined task signal or software interrupt (see the "Exec Interrupts" chapter).

Like many Exec structures, ports may be given a symbolic name. Such names are particularly useful for tasks that must rendezvous with dynamically created ports. They are also useful for debugging purposes.

A message port consists of a **MsgPort** structure as defined in the *<exec/ports.h>* and *<exec/ports.i>* include files. The C structure for a port is as follows:

```
struct MsgPort {
    struct Node  mp_Node;
    UBYTE        mp_Flags;
    UBYTE        mp_SigBit;
    struct Task *mp_SigTask;
    struct List  mp_MsgList;
};
```

mp_Node
is a standard **Node** structure. This is useful for tasks that might want to rendezvous with a particular message port by name.

mp_Flags
are used to indicate message arrival actions. See the explanation below.

mp_SigBit
is the signal bit *number* when a port is used with the task signal arrival action.

mp_SigTask
is a pointer to the task to be signaled. If a software interrupt arrival action is specified, this is a pointer to the interrupt structure.

mp_MsgList
is the list header for all messages queued to this port. (See the ''Exec Lists and Queues'' chapter).

The **mp_Flags** field contains a subfield indicated by the **PF_ACTION** mask. This sub-field specifies the message arrival action that occurs when a port receives a new message. The possibilities are as follows:

PA_SIGNAL
This flag tells Exec to signal the **mp_SigTask** using signal number **mp_SigBit** on the arrival of a new message. Every time a message is put to the port another signal will occur regardless of how many messages have been queued to the port.

PA_SOFTINT
This flag tells Exec to **Cause()** a software interrupt when a message arrives at the port. In this case, the **mp_SigTask** field must contain a pointer to a **struct Interrupt** rather than a Task pointer. The software interrupt will be Caused every time a message is received.

PA_IGNORE
This flag tells Exec to perform no operation other than queuing the message. This action is often used to stop signaling or software interrupts without disturbing the contents of the **mp_SigTask** field.

It is important to realize that a port's arrival action will occur for each new message queued, and that there is not a one-to-one correspondence between messages and signals. Task signals are only single-bit flags so there is no record of how many times a particular signal occurred. There may be many messages queued and only a single task signal; sometimes however there may be a signal, but no messages. All of this has certain implications when designing code that deals with these actions. Your code should not depend on receiving a signal for every message at your port. All of this is also true for software interrupts.

CREATING A MESSAGE PORT

To create a new message port using an operating system release prior to V36, you must allocate and initialize a **MsgPort** structure. If you want to make the port *public*, you will also need to call the **AddPort()** function. Don't make a port public when it is not necessary for it to be so. The easiest way to create a port is to use the amiga.lib function **CreatePort(name,priority)**. If NULL is passed for the name, the port will not be made public. Port structure initialization involves setting up a **Node** structure, establishing the message arrival action with its parameters, and initializing the list header. The following example of port creation is equivalent to the **CreatePort()** function as supplied in *amiga.lib*:

```
struct MsgPort *CreatePort(UBYTE *name, LONG pri)
{
    LONG sigBit;
    struct MsgPort *mp;

    if ((sigBit = AllocSignal(-1L)) == -1) return(NULL);

    mp = (struct MsgPort *) AllocMem((ULONG)sizeof(struct MsgPort), (ULONG)MEMF_PUBLIC | MEMF_CLEAR);
    if (!mp) {
        FreeSignal(sigBit);
        return(NULL);
    }
    mp->mp_Node.ln_Name = name;
    mp->mp_Node.ln_Pri  = pri;
    mp->mp_Node.ln_Type = NT_MSGPORT;
    mp->mp_Flags        = PA_SIGNAL;
    mp->mp_SigBit       = sigBit;
    mp->mp_SigTask      = (struct Task *)FindTask(0L);    /* Find THIS task. */

    if (name) AddPort(mp);
    else NewList(&(mp->mp_MsgList));                     /* init message list */

    return(mp);
}
```

As of V36 the Exec **CreateMsgPort()** function can be used to create a message port. This function allocates and initializes a new message port. Just like **CreatePort()**, a signal bit will be allocated and the port will be initialized to signal the creating task (mp_SigTask) when a message arrives at this port. To make the port public after **CreateMsgPort()**, you must fill out the ln_Name field and call **AddPort()**. If you do this, you must remember to **RemPort()** the port from the public list in your cleanup. If you need to create a message port and your application already requires Release 2 or greater, you can use **CreateMsgPort()** instead of **CreatePort()**. The following is an example of the usage of **CreateMsgPort()**.

```
struct MsgPort *newmp;
                            /* A private message port has been created. CreateMsgPort() */
if (newmp = CreateMsgPort()) /* returns NULL if the creation of the message port failed. */
{
    newmp->mp_Node.ln_Name = "Griffin";
    newmp->mp_Node.ln_Pri  = 0;          /* To make it public fill in the fields */
    AddPort(newmp);                      /* with appropriate values.             */
}
```

DELETING A MESSAGE PORT

Before a message port is deleted, all outstanding messages from other tasks must be returned. This is done by getting and replying to all messages at the port until message queue is empty. Of course, there is no need to reply to messages owned by the current task (the task performing the port deletion). Public ports attached to the system with **AddPort()** must be removed from the system with **RemPort()** before deallocation. This *amiga.lib* functions **CreatePort()** and **DeletePort()** handle this automatically.

The following example of port deletion is equivalent to the **DeletePort()** function as supplied in *amiga.lib*. Note that **DeletePort()** must only be used on ports created with **CreatePort()**.

```
void DeletePort(mp)
struct MsgPort *mp;
{
    if ( mp->mp_Node.ln_Name ) RemPort(mp);      /* if it was public... */

    mp->mp_SigTask         = (struct Task *) -1; /* Make it difficult to re-use the port */
    mp->mp_MsgList.lh_Head = (struct Node *) -1;

    FreeSignal( mp->mp_SigBit );
    FreeMem( mp, (ULONG)sizeof(struct MsgPort) );
}
```

To delete ports created with **CreateMsgPort()**, **DeleteMsgPort()** must be used. Note that these functions are only available in V36 and higher. If the port was made public with **AddPort()**, **RemPort()** must be used first, to remove the port from the system. Again, make sure all outstanding messages are replied to, so that the message queue is empty.

```
struct MsgPort *newmp;

if (newmp)
{
    if ( newmp->mp_Node.ln_Name ) RemPort(newmp);    /* if it was public... */
    DeleteMsgPort(newmp);
}
```

HOW TO RENDEZVOUS AT A MESSAGE PORT

The **FindPort()** function provides a means of finding the address of a public port given its symbolic name. For example, **FindPort("Griffin")** will return either the address of the message port named "Griffin" or NULL indicating that no such public port exists. Since **FindPort()** does not do any arbitration over access to public ports, the usage of **FindPort()** must be protected with **Forbid()/Permit()**. Names should be unique to prevent collisions among multiple applications. It is a good idea to use your application name as a prefix for your port name. **FindPort()** does not arbitrate for access to the port list. The owner of a port might remove it at any time. For these reasons a **Forbid()/Permit()** pair is required for the use of **FindPort()**. The port address can no longer be regarded as being valid after **Permit()** unless your application knows that the port cannot go away (for example, if your application created the port).

The following is an example of how to safely put a message to a specific port:

```
#include <exec/types.h>
#include <exec/ports.h>

BOOL MsgPort SafePutToPort(struct Message *message, STRPTR portname)
{
    struct MsgPort *port;

    Forbid();
    port = FindPort(portname);
    if (port) PutMsg(port,message);
    Permit();
    return(port ? TRUE : FALSE);      /* If FALSE, the port was not found */

    /* Once we've done a Permit(), the port might go away and leave us with an invalid port    */
    /* address. So we return just a BOOL to indicate whether the message has been sent or not. */
}
```

Messages

As mentioned earlier, a message contains both system header information and the actual message content. The system header is of the **Message** form defined in *<exec/ports.h>* and *<exec/ports.i>*. In C this structure is as follows:

```
struct Message {
    struct Node     mn_Node;
    struct MsgPort *mn_ReplyPort;
    UWORD           mn_Length;
};
```

mn_Node
: is a standard **Node** structure used for port linkage.

mn_ReplyPort
: is used to indicate a port to which this message will be returned when a reply is necessary.

mn_Length
: indicates the total length of the message, including the Message structure itself.

This structure is always attached to the head of all messages. For example, if you want a message structure that contains the x and y coordinates of a point on the screen, you could define it as follows:

```
struct XYMessage {
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
}
```

For this structure, the **mn_Length** field should be set to sizeof(struct **XYMessage**).

PUTTING A MESSAGE

A message is delivered to a given destination port with the **PutMsg()** function. The message is queued to the port, and that port's arrival action is invoked. If the action specifies a task signal or a software interrupt, the originating task may temporarily lose the processor while the destination processes the message. If a reply to the message is required, the **mn_ReplyPort** field must be set up prior to the call to **PutMsg()**.

Here is a code fragment for putting a message to a public port. A complete example is printed at the end of the chapter.

```
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/ports.h>
#include <libraries/dos.h>


VOID main(VOID);
BOOL SafePutToPort(struct Message *, STRPTR);

struct XYMessage {
    struct Message xy_Msg;
    UWORD          xy_X;
    UWORD          xy_Y;
};
```

```
VOID main(VOID)
{
    struct MsgPort *xyport, *xyreplyport;
    struct XYMessage *xymsg, *msg;
    BOOL   foundport;

    /* Allocate memory for the message we're going to send. */
    if (xymsg = (struct XYMessage *) AllocMem(sizeof(struct XYMessage),MEMF_PUBLIC | MEMF_CLEAR))
    {

        /* The replyport we'll use to get response */
        if (xyreplyport = CreateMsgPort()) {  /* or use CreatePort(0,0) */


            xymsg->xy_Msg.mn_Node.ln_Type = NT_MESSAGE;          /* Compose the message */
            xymsg->xy_Msg.mn_Length = sizeof(struct XYMessage);
            xymsg->xy_Msg.mn_ReplyPort = xyreplyport;
            xymsg->xy_X = 10;
            xymsg->xy_Y = 20;

            /* Now try to send that message to a public port named "xyport".
             * If foundport eq 0, the port isn't out there.
             */
            if (foundport = SafePutToPort((struct Message *)xymsg, "xyport"))
            {

            . . .                           /* Now let's wait till the someone responds... */

            }
            else printf("Couldn't find 'xyport'\n");

            DeleteMsgPort(xyreplyport);                    /* Use DeletePort() if the port was */
        }                                                  /* was created with CreatePort().   */
        else printf("Couldn't create message port\n");
        FreeMem(xymsg, sizeof(struct XYMessage));
    }
    else printf("Couldn't get memory for xymessage\n");
}
```

WAITING FOR A MESSAGE

A task may go to sleep waiting for a message to arrive at one or more ports. This technique is widely used on the Amiga as a general form of event notification. For example, it is used extensively by tasks for I/O request completion.

The **MsgPort.mp_SigTask** field contains the address of the task to be signaled and **mp_SigBit** contains a preallocated signal number (as described in the "Exec Tasks" chapter).

You can call the **WaitPort()** function to wait for a message to arrive at a port. This function will return the first message (it may not be the only) queued to a port. Note that your application must still call **GetMsg()** to remove the message from the port. If the port is empty, your task will go to sleep waiting for the first message. If the port is not empty, your task will not go to sleep. It is possible to receive a signal for a port without a message being present yet. The code processing the messages should be able to handle this. The following code illustrates **WaitPort()**.

```
struct XYMessage *xy_msg;
struct MsgPort   *xyport;

xyport = CreatePort("xyport", 0);
if (xyport == 0)
{
    printf("Couldn't create xyport\n");
    exit(31);
}

xy_msg = WaitPort(xyport);     /* go to sleep until message arrives */
```

A more general form of waiting for a message involves the use of the **Wait()** function (see the "Exec Signals" chapter). This function waits for task event signals directly. If the signal assigned to the message port occurs, the task will awaken. Using the **Wait()** function is more general because you can wait for more than one signal. By combining the signal bits from each port into one mask for the **Wait()** function, a loop can be set up to process all messages at all ports.

Here's an example using **Wait()**:

```
struct XYMessage *xy_msg;
struct MsgPort   *xyport;
ULONG usersig, portsig;
BOOL ABORT = FALSE;

if (xyport = CreatePort("xyport", 0))
{
    portsig = 1 << xyport->mp_SigBit;
    usersig = SIGBREAKF_CTRL_C;          /* User can break with CTRL-C.  */
    for (;;)
    {
        signal = Wait(portsig | usersig);  /* Sleep till someone signals.  */

        if (signal & portsig)            /* Got a signal at the msgport. */
        {   . . .
        }
        if (signal & usersig)            /* Got a signal from the user.  */
        {
            ABORT = TRUE;                /* Time to clean up.            */
             . . .
        }
        if (ABORT) break;
    }
    DeletePort(xyport);
}
else printf("Couldn't create xyport\n");
```

> *WaitPort() Does Not Remove A Message.* **WaitPort()** only returns a pointer to the first message in a port. It does not actually remove the message from the port queue.

GETTING A MESSAGE

Messages are usually removed from ports with the **GetMsg()** function. This function removes the next message at the head of the port queue and returns a pointer to it. If there are no messages in a port, this function returns a zero.

The example below illustrates the use of **GetMsg()** to print the contents of all messages in a port:

```
while (xymsg = GetMsg(xyport)) printf("x=%ld y=%ld\n", xymsg->xy_X, xymsg->xy_Y);
```

Certain messages may be more important than others. Because ports impose FIFO ordering, these important messages may get queued behind other messages regardless of their priority. If it is necessary to recognize more important messages, it is easiest to create another port for these special messages.

REPLYING

When the operations associated with receiving a new message are finished, it is usually necessary to send the message back to the originator. The receiver replies the message by returning it to the originator using the **ReplyMsg()** function. This is important because it notifies the originator that the message can be reused or deallocated. The **ReplyMsg()** function serves this purpose. It returns the message to the port specified in the **mn_ReplyPort** field of the message. If this field is zero, no reply is returned.

The previous example can be enhanced to reply to each of its messages:

```
while (xymsg = GetMsg(xyport)) {
    printf("x=%ld y=%ld\n", xymsg->xy_X, xymsg->xy_Y);
    ReplyMsg(xymsg);
}
```

Notice that the reply does not occur until *after* the message values have been used.

Often the operations associated with receiving a message involve returning ***results*** to the originator. Typically this is done within the message itself. The receiver places the results in fields defined (or perhaps reused) within the message body before replying the message back to the originator. Receipt of the replied message at the originator's reply port indicates it is once again safe for the originator to use or change the values found within the message.

The following are two short example tasks that communicate by sending, waiting for and replying to messages. Run these two programs together.

Port1.c

```
;/* port1.c - Execute me to compile with SAS C 5.10
LC -b1 -cfistq -v -y -j73 port1.c
Blink FROM LIB:c.o,port1.o TO port1 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

* port1.c - port and message example, run at the same time as port2.c
*/

#include <exec/types.h>
#include <exec/ports.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL-C handling */
int chkabort(void) {return(0);}
#endif

struct XYMessage {
    struct Message xym_Msg;
    WORD           xy_X;
    WORD           xy_Y;
};

void main(int argc, char **argv)
{
    struct MsgPort *xyport;
    struct XYMessage *xymsg;
    ULONG portsig, usersig, signal;
    BOOL ABORT = FALSE;

    if (xyport = CreatePort("xyport", 0))
    {
        portsig = 1 << xyport->mp_SigBit;       /* Give user a 'break' signal. */
        usersig = SIGBREAKF_CTRL_C;

        printf("Start port2 in another shell.  CTRL-C here when done.\n");
        do
        {                                               /* port1 will wait forever and reply   */
            signal = Wait(portsig | usersig);           /* to messages, until the user breaks. */

                                  /* Since we only have one port that might get messages we    */
            if (signal & portsig)  /* have to reply to, it is not really necessary to test for  */
            {                     /* the portsignal. If there is not message at the port, xymsg */
```

```
            while(xymsg = (struct XYMessage *)GetMsg(xyport))        /* simply will be NULL. */
            {
                printf("port1 received: x = %d y = %d\n", xymsg->xy_X, xymsg->xy_Y);

                xymsg->xy_X += 50;       /* Since we have not replied yet to the owner of    */
                xymsg->xy_Y += 50;       /* xymsg, we can change the data contents of xymsg. */

                printf("port1 replying with: x = %d y = %d\n", xymsg->xy_X, xymsg->xy_Y);
                ReplyMsg((struct Message *)xymsg);
            }
        }

        if (signal & usersig)                                    /* The user wants to abort. */
        {
            while(xymsg = (struct XYMessage *)GetMsg(xyport))    /* Make sure port is empty. */
                ReplyMsg((struct Message *)xymsg);
            ABORT = TRUE;
        }
    }
    while (ABORT == FALSE);
        DeletePort(xyport);
    }
    else printf("Couldn't create 'xyport'\n");
}
```

Port2.c

```
;/* port2.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 port2.c
Blink FROM LIB:c.o,port2.o TO port2 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** port2.c - port and message example, run at the same time as port1.c
*/

#include <exec/types.h>
#include <exec/ports.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL-C handling */
int chkabort(void) {return(0);}
#endif

BOOL SafePutToPort(struct Message *, STRPTR);

struct XYMessage {
    struct Message xy_Msg;
    WORD           xy_X;
    WORD           xy_Y;
};

void main(int argc, char **argv)
{
    struct MsgPort *xyreplyport;
    struct XYMessage *xymsg, *reply;
                                                   /* Using CreatePort() with no name       */
    if (xyreplyport = CreatePort(0,0))             /* because this port need not be public. */
    {
        if (xymsg = (struct XYMessage *) AllocMem(sizeof(struct XYMessage), MEMF_PUBLIC | MEMF_CLEAR))
        {
            xymsg->xy_Msg.mn_Node.ln_Type = NT_MESSAGE;            /* make up a message,        */
            xymsg->xy_Msg.mn_Length = sizeof(struct XYMessage);    /* including the reply port. */
            xymsg->xy_Msg.mn_ReplyPort = xyreplyport;
            xymsg->xy_X = 10;                               /* our special message information. */
            xymsg->xy_Y = 20;

            printf("Sending to port1: x = %d y = %d\n", xymsg->xy_X, xymsg->xy_Y);
                                                            /* port2 will simply try to put  */
```

```
        if (SafePutToPort((struct Message *)xymsg, "xyport"))  /* one message to port1 wait for */
        {                                                      /*  the reply, and then exit     */
            WaitPort(xyreplyport);
            if (reply = (struct XYMessage *)GetMsg(xyreplyport))
                printf("Reply contains: x = %d y = %d\n",          /* We don't ReplyMsg since   */
                        xymsg->xy_X, xymsg->xy_Y);                  /* WE initiated the message. */

                  /* Since we only use this private port for receiving replies, and we sent     */
                  /* only one and got one reply there is no need to cleanup. For a public port, */
                  /* or if you pass a pointer to the port to another process, it is a very good */
                  /* habit to always handle all messages at the port before you delete it.     */
        }
        else printf("Can't find 'xyport'; start port1 in a separate shell\n");
        FreeMem(xymsg, sizeof(struct XYMessage));
    }
    else printf("Couldn't get memory\n");
    DeletePort(xyreplyport);
  }
  else printf("Couldn't create xyreplyport\n");
}


BOOL SafePutToPort(struct Message *message, STRPTR portname)
{
    struct MsgPort *port;

    Forbid();
    port = FindPort(portname);
    if (port) PutMsg(port, message);
    Permit();
    return(port ? TRUE : FALSE); /* FALSE if the port was not found */

        /* Once we've done a Permit(), the port might go away and leave us with an invalid port    */
}       /* address. So we return just a BOOL to indicate whether the message has been sent or not. */
```

Function Reference

The following chart gives a brief description of the Exec functions that control inter-task communication with messages and ports. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 24-1: Exec Message and Port Functions

Function	Description
AddPort()	Add a public message port to the system list.
CreateMsgPort()	Allocate and initialize a new message port (V37).
DeleteMsgPort()	Free a message port, created with **CreateMsgPort()** (V37).
FindPort()	Find a public message port in the system list.
GetMsg()	Get next message from the message port.
PutMsg()	Put a message to a message port.
RemPort()	Remove a message port from the system list.
ReplyMsg()	Reply to a message on its reply port.

Table 24-2: Amiga.lib Exec Support Functions

Function	Description
CreatePort()	Allocate and initialize a new message port, make public if named
DeletePort()	Delete a message port, created with **CreatePort()**.

Chapter 25
EXEC SEMAPHORES

Semaphores are a feature of Exec which provide a general method for tasks to arbitrate for the use of memory or other system resources they may be sharing. This chapter describes the structure of Exec semaphores and the various support functions provided for their use. Since the semaphore system uses Exec lists and signals, some familiarity with these concepts is helpful for understanding semaphores.

In any multitasking or multi-processing system there is a need to share data among independently executing tasks. If the data is static (that is, it never changes), then there is no problem. However, if the data is variable, then there must be some way for a task that is about to make a change to keep other tasks from looking at the data.

For example, to add a node to a linked list of data, a task would normally just add the node. However, if the list is shared with other tasks, this could be dangerous. Another task could be walking down the list while the change is being made and pick up an incorrect pointer. The problem is worse if two tasks attempt to add an item to the list at the same time. Exec semaphores provide a way to prevent such problems.

A semaphore is much like getting a key to a locked data item. When you have the key (semaphore), you can access the data item without worrying about other tasks causing problems. Any other tasks that try to obtain the semaphore will be put to sleep until the semaphore becomes available. When you have completed your work with the data, you return the semaphore.

For semaphores to work correctly, there are two restrictions that *must* be observed at all times:

1) All tasks using shared data that is protected by a semaphore must *always ask for the semaphore first before accessing the data.* If some task accesses the data directly without first going through the semaphore, the data may be corrupted. No task will have safe access to the data.

2) A deadlock will occur if a task that owns an exclusive semaphore on some data inadvertently calls another task which tries to get an exclusive semaphore on that same data in blocking mode. Deadlocks and other such issues are beyond the scope of this manual. For more details on deadlocks and other problems of shared data in a multitasking system and the methods used to prevent them, refer to a textbook in computer science such as *Operating Systems* by Tannenbaum (Prentice-Hall).

Semaphore Functions

Exec provides a variety of useful functions for setting, checking and freeing semaphores. The prototypes for these functions are as follows.

```
VOID  AddSemaphore ( struct SignalSemaphore *sigSem );
ULONG AttemptSemaphore( struct SignalSemaphore *sigSem );
struct SignalSemaphore *FindSemaphore( UBYTE *sigSem );
VOID InitSemaphore( struct SignalSemaphore *sigSem );

VOID ObtainSemaphore( struct SignalSemaphore *sigSem );
VOID ObtainSemaphoreList( struct List *sigSem );
void ObtainSemaphoreShared( struct SignalSemaphore *sigSem );

VOID ReleaseSemaphore( struct SignalSemaphore *sigSem );
VOID ReleaseSemaphoreList( struct List *sigSem );
VOID RemSemaphore( struct SignalSemaphore *sigSem );
```

THE SIGNAL SEMAPHORE

Exec semaphores are signal based. Using signal semaphores is the easiest way to protect shared, single-access resources in the Amiga. Your task will sleep until the semaphore is available for use. The **SignalSemaphore** structure is as follows:

```
struct SignalSemaphore {
    struct  Node ss_Link;
    SHORT   ss_NestCount;
    struct  MinList ss_WaitQueue;
    struct  SemaphoreRequest ss_MultipleLink;
    struct  Task *ss_Owner;
    SHORT   ss_QueueCount;
};
```

ss_Link
: is the node structure used to link semaphores together. The ln_Pri and ln_Name fields are used to set the priority of the semaphore in a list and to name the semaphore for public access. If a semaphore is not public the **ln_Name** and **ln_Pri** fields may be left NULL.

ss_NestCount
: is the count of number of locks the current owner has on the semaphore.

ss_WaitQueue
: is the **List** header for the list of other tasks waiting for this semaphore.

ss_MultipleLink
: is the SemaphoreRequest used by **ObtainSemaphoreList()**.

ss_Owner
: is the pointer to the current owning task.

ss_QueueCount
: is the number of other tasks waiting for the semaphore.

A practical application of a **SignalSemaphore** would be to use it as the base of a shared data structure. For example:

```
struct SharedList {
    struct SignalSemaphore sl_Semaphore;
    struct MinList         sl_List;
};
```

Creating a SignalSemaphore Structure

To initialize a **SignalSemaphore** structure use the **InitSemaphore()** function. This function initializes the list structure and the nesting and queue counters. It does not change the semaphore's name or priority fields.

This fragment creates and initializes a semaphore for a data item such as the **SharedList** structure above.

```
struct SharedList *slist;

if (slist=(struct SharedList *)
    AllocMem(sizeof(struct SharedList),MEMF_PUBLIC|MEMF_CLEAR))
{
    NewList(&slist->sl_List);                          /* Initialize the MinList */
    InitSemaphore((struct SignalSemaphore *)slist);  /* And initialize the semaphore */

    /* The semaphore can now be used. */
}
else printf("Can't allocate structure\n");
```

Making a SignalSemaphore Available to the Public

A semaphore should be used internally in your program if it has more than one task operating on shared data structures. There may also be cases when you wish to make a data item public to other applications but still need to restrict its access via semaphores. In that case, you would give your semaphore a unique name and add it to the public **SignalSemaphore** list maintained by Exec. The **AddSemaphore()** function does this for you. This works in a manner similar to **AddPort()** for message ports.

To create and initialize a *public* semaphore for a data item and add it to the public semaphore list maintained by Exec, the following function should be used. (This will prevent the semaphore from being added or removed more than once by separate programs that use the semaphore).

```
UBYTE *name;   /* name of semaphore to add */
struct SignalSemaphore *semaphore;

Forbid();
/* Make sure the semaphore name is unique */
if (!FindSemaphore(name)) {
    /* Allocate memory for the structure */
    if (sema=(struct SignalSemaphore *)
        AllocMem(sizeof(struct SignalSemaphore),MEMF_PUBLIC|MEMF_CLEAR))
    {
        sema->ss_Link.ln_Pri=0;          /* Set the priority to zero */
        sema->ss_Link.ln_Name=name;
        /* Note that the string 'name' is not copied. If that is needed, allocate memory */
        /* for it and copy the string. And add the semaphore the the system list */
        AddSemaphore(semaphore);
    }
}
Permit();
```

A value of NULL for **semaphore** means that the semaphore already exists or that there was not enough free memory to create it.

Before using the data item or other resource which is protected by a semaphore, you must first obtain the semaphore. Depending on your needs, you can get either *exclusive* or *shared* access to the semaphore.

Obtaining a SignalSemaphore Exclusively

The **ObtainSemaphore()** function can be used to get an exclusive lock on a semaphore. If another task currently has an exclusive or shared lock(s) on the semaphore, your task will be put to sleep until all locks on the the semaphore are released.

> *Semaphore Nesting.* **SignalSemaphores** have nesting. That is, if your task already owns the semaphore, it will get a second ownership of that semaphore. This simplifies the writing of routines that must own the semaphore but do not know if the caller has obtained it yet.

To obtain a semaphore use:

```
struct SignalSemaphore *semaphore;
ObtainSemaphore(semaphore);
```

To get an exclusive lock on a *public* semaphore, the following code should be used:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();     /* Make sure the semaphore will not go away if found. */
if (semaphore=FindSemaphore(name))
    ObtainSemaphore(semaphore);
Permit();
```

The value of **semaphore** is NULL if the semaphore does not exist. This is *only* needed if the semaphore has a chance of going away at any time (i.e., the semaphore is public and might be removed by some other program). If there is a guarantee that the semaphore will not disappear, the semaphore address could be cached, and all that would be needed is a call to the **ObtainSemaphore()** function.

Obtaining a Shared SignalSemaphore

For read-only purposes, multiple tasks may have a shared lock on a signal semaphore. If a semaphore is already exclusively locked, all attempts to obtain the semaphore shared will be blocked until the exclusive lock is released. At that point, all shared locks will be obtained and the calling tasks will wake up.

To obtain a shared semaphore, use:

```
struct SignalSemaphore *semaphore;
ObtainSemaphoreShared(semaphore);
```

To obtain a *public* shared semaphore, the following code should be used:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();
if (semaphore = FindSemaphore(name))
    ObtainSemaphoreShared(semaphore);
Permit();
```

Checking a SignalSemaphore

When you attempt to obtain a semaphore with **ObtainSemaphore()**, your task will be put to sleep if the semaphore is not currently available. If you do not want to wait, you can call **AttemptSemaphore()** instead. If the semaphore is available for exclusive locking, **AttemptSemaphore()** obtains it for you and returns TRUE. If it is not available, the function returns FALSE immediately instead of waiting for the semaphore to be released.

To attempt to obtain a semaphore, use the following:

```
struct SignalSemaphore *semaphore;
AttemptSemaphore(semaphore);
```

To make an attempt to obtain a *public* semaphore, the following code should be used:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();
if (semaphore = FindSemaphore(name)) AttemptSemaphore(semaphore);
Permit();
```

Releasing a SignalSemaphore

Once you have obtained the semaphore and completed any operations on the semaphore protected object, you should release the semaphore. The **ReleaseSemaphore()** function does this. For each successful **ObtainSemaphore()**, **ObtainSemaphoreShared()** and **AttemptSemaphore()** call you make, you *must* have a matching **ReleaseSemaphore()** call.

Removing a SignalSemaphore Structure

Semaphore resources can only be freed if the semaphore is not locked. A public semaphore should first be removed from the system semaphore list with the **RemSemaphore()** function. This prevents other tasks from finding the semaphore and trying to lock it. Once the semaphore is removed from the system list, the semaphore should be locked exclusively so no other task can lock it. Once the lock is obtained, it can be released again, and the resources can be deallocated.

The following code should be used to remove a public semaphore:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();
if (semaphore=FindSemaphore(name))
{
    RemSemaphore(semaphore);       /* So no one else can find it... */
    ObtainSemaphore(semaphore);    /* Wait for us to be last user...*/
    ReleaseSemaphore(semaphore);   /* Ready for cleanup...          */
}
FreeMem(semaphore, sizeof(struct SignalSemaphore));
Permit();
```

MULTIPLE SEMAPHORES

The semaphore system has the ability to ask for ownership of a complete list of semaphores. This can help prevent deadlocks when there are two or more tasks trying to get the same set of semaphores. If task A gets semaphore 1 and tries to obtain semaphore 2 *after* task B has obtained semaphore 2 but *before* task B tries to obtain semaphore 1 then both tasks will hang. Exec provides **ObtainSemaphoreList()** and **ReleaseSemaphoreList()** to prevent this problem.

A semaphore list is a list header to a list that contains **SignalSemaphore** structures. The semaphore list must not contain any public semaphores. This is because the semaphore list functions use the standard node structures in the semaphore.

To arbitrate access to a semaphore list use another semaphore. Create a public semaphore and use it to arbitrate access to the list header of the semaphore list. This also gives you a locking semaphore, protecting the **ObtainSemaphoreList()** call. Once you have gained access to the list with **ObtainSemaphore()**, you may obtain all the semaphores on the list via **ObtainSemaphoreList()** (or get individual semaphores with **ObtainSemaphore()**). When you are finished with the protected objects, release the semaphores on the list with **ReleaseSemaphoreList()**, and then release the list semaphore via **ReleaseSemaphore()**.

For example:

```
ObtainSemaphore((struct SignalSemaphore *)SemaphoreList);
ObtainSemaphoreList(SemaphoreList->sl_List);

/* At this point the objects are protected, and can be manipulated */

ReleaseSemaphoreList(SemaphoreList->sl_List);
ReleaseSemaphore((struct SignalSemaphore *)SemaphoreList);
```

See the **SharedList** structure above for an example of a semaphore structure with a list header.

SEMAPHORE EXAMPLE

A simple "do nothing" example of Exec signal semaphore use is shown below. When the semaphore is owned by a task, attempted access by other tasks will block. A nesting count is maintained, so the current task can safely call **ObtainSemaphore()** on the same semaphore.

```
/* semaphore.c - Exec semaphore example - compile with lc -L semaphore.c */
#include <exec/types.h>
#include <exec/semaphores.h>
#include <clib/exec_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)   { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

struct SignalSemaphore LockSemaphore = {0};

VOID main(int argc,char *argv[])
{
    InitSemaphore(&LockSemaphore);
    ObtainSemaphore(&LockSemaphore);  /* Task now owns the semaphore. */
        ...
    ReleaseSemaphore(&LockSemaphore); /* Task has released the semaphore. */
}
```

Function Reference

The following charts give a brief description of the Exec semaphore functions. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 25-1: Exec Semaphore Functions

Exec Semaphore Function	Description
AddSemaphore()	Initialize and add a signal semaphore to the system.
AttemptSemaphore()	Try to get an exclusive lock on a signal semaphore without blocking.
FindSemaphore()	Find a given system signal semaphore.
InitSemaphore()	Initialize a signal semaphore.
ObtainSemaphore()	Try to get exclusive access to a signal semaphore.
ObtainSemaphoreList()	Try to get exclusive access to a list of signal semaphores.
ObtainSemaphoreShared()	Try to get shared access to a signal semaphore (V36).
ReleaseSemaphore()	Release the lock on a signal semaphore.
ReleaseSemaphoreList()	Release the locks on a list of signal semaphores.
RemSemaphore()	Remove a signal semaphore from the system.

Chapter 26
EXEC INTERRUPTS

Exec manages the decoding, dispatching, and sharing of all system interrupts. This includes control of hardware interrupts, software interrupts, task-relative interrupts (see the discussion of exceptions in the ''Exec Tasks'' chapter), and interrupt disabling and enabling. In addition, Exec supports a more extended prioritization of interrupts than that provided in the 68000.

The proper operation of multitasking depends heavily on the consistent management of the interrupt system. Task activities are often driven by intersystem communication that is originated by various interrupts.

SEQUENCE OF EVENTS DURING AN INTERRUPT

Before useful interrupt handling code can be executed, a considerable amount of hardware and software activity must occur. Each interrupt must propagate through several hardware and software interfaces before application code is finally dispatched:

- A hardware device decides to cause an interrupt and sends a signal to the interrupt control portions of the 4703 (Paula) custom chip.

- The 4703 interrupt control logic notices this new signal and performs two primary operations. First, it records that the interrupt has been requested by setting a flag bit in the **INTREQ** register. Second, it examines the **INTENA** register to determine whether the corresponding interrupt and the interrupt master are enabled. If both are enabled, the 4703 generates an interrupt request by placing the priority level of the request onto the three 68000 interrupt control input lines (**IPL0**, **IPL1**, **IPL2**).

- These three signals correspond to seven interrupt priority levels in the 68000. If the priority of the new interrupt is *greater* than the current processor priority, an interrupt sequence is initiated. The priority level of the new interrupt is used to index into the top seven words of the processor address space. The odd byte (a vector number) of the indexed word is fetched and then shifted left by two to create an offset into the processor's auto-vector interrupt table. The vector offsets used are in the range of $064 to $07C. These are labeled as *interrupt autovectors* in the 68000 manual. The auto-vector table appears in low memory on a 68000 system, but its location for other 68000 family processors is determined by the processor's CPU Vector Base Register (**VBR**). **VBR** can be accessed from supervisor mode with the **MOVEC** instruction.

- The processor then switches into *supervisor* mode (if it is not already in that mode), and saves copies of the status register and program counter (**PC**) onto the top of the *system* stack (additional information may be saved by processors other than the 68000). The processor priority is then raised to the level of the active interrupt.

- From the low memory vector address (calculated in step three above), a 32-bit *autovector* address is fetched and loaded into the program counter. This is an entry point into Exec's interrupt dispatcher.

- Exec must now further decode the interrupt by examining the **INTREQ** and **INTENA** 4703 chip registers. Once the active interrupt has been determined, Exec indexes into an **ExecBase** array to fetch the interrupt's handler entry point and handler data pointer addresses.

- Exec now turns control over to the interrupt handler by calling it as if it were a subroutine. This handler may deal with the interrupt directly or may propagate control further by invoking interrupt server chain processing.

You can see from the above discussion that the interrupt autovectors *should never be altered by the user*. If you wish to provide your own system interrupt handler, you must use the Exec **SetIntVector()** function. You should not change the contents of any autovector location.

Task multiplexing usually occurs as the result of an interrupt. When an interrupt has finished and the processor is about to return to user mode, Exec determines whether task-scheduling attention is required. If a task was signaled during interrupt processing, the task scheduler will be invoked. Because Exec uses preemptive task scheduling, it can be said that the interrupt subsystem is the heart of task multiplexing. If, for some reason, interrupts do not occur, a task might execute forever because it cannot be forced to relinquish the CPU.

Table 26-1: Interrupts by Priority

Hardware Priority	Exec Pseudo-Priority	Description	Label	Type
1	1	Serial Transmit Buffer Empty	TBE	H
	2	Disk Block Complete	DSKBLK	H
	3	Software Interrupt	SOFTINT	H
2	4	External INT2 & CIAA	PORTS	S
3	5	Graphics Coprocessor	COPER	S
	6	Vertical Blank Interval	VERTB	S
	7	Blitter Finished	BLIT	H
4	8	Audio Channel 2	AUD2	H
	9	Audio Channel 0	AUD0	H
	10	Audio Channel 3	AUD3	H
	11	Audio Channel 1	AUD1	H
5	12	Serial Receive Buffer Full	RBF	H
	13	Disk Sync Pattern Found	DSKSYNC	H
6	14	External INT6 & CIAB	EXTER	S
	15	Special (Master Enable)	INTEN	–
7	–	Non-Maskable Interrupt	NMI	S

INTERRUPT PRIORITIES

Interrupts are prioritized in hardware and software. The 68000 CPU priority at which an interrupt executes is determined strictly by hardware. In addition to this, the software imposes a finer level of *pseudo-priorities* on interrupts with the same CPU priority. These pseudo-priorities determine the order in which simultaneous interrupts of the same CPU priority are processed. Multiple interrupts with the same CPU priority but a different pseudo-priority will not interrupt one another. Interrupts are serviced by either an exclusive handler or by server chains to which many servers may be attached, as shown in the **Type** field of the next table. The table above summarizes all interrupts by priority.

The 8520s (also called CIAs) are Amiga peripheral interface adapter chips that generate the **INT2** and **INT6** interrupts. For more information about them, see the *Amiga Hardware Reference Manual.*

As described in the Motorola 68000 programmer's manual, interrupts may nest only in the direction of higher priority. Because of the time-critical nature of many interrupts on the Amiga, the CPU priority level *must never be changed* by user or system code. When the system is running in user mode (multitasking), the CPU priority level must remain set at zero. When an interrupt occurs, the CPU priority is raised to the level appropriate for that interrupt. Lowering the CPU priority would permit unlimited interrupt recursion on the system stack and would ''short-circuit'' the interrupt-priority scheme.

Because it is dangerous on the Amiga to hold off interrupts for any period of time, higher-level interrupt code must perform its business and exit promptly. If it is necessary to perform a time-consuming operation as the result of a high-priority interrupt, the operation should be deferred either by posting a *software interrupt* or by signalling a task. In this way, interrupt response time is kept to a minimum. Software interrupts are described in a later section.

NONMASKABLE INTERRUPT

The 68000 provides a nonmaskable interrupt (**NMI**) of CPU priority 7. Although this interrupt cannot be generated by the Amiga hardware itself, it can be generated on the expansion bus by external hardware. Because this interrupt does not pass through the 4703 interrupt controller circuitry, it is capable of violating system code critical sections. In particular, it short-circuits the **DISABLE** mutual-exclusion mechanism. Code that uses **NMI** must not assume that it can access system data structures.

Servicing Interrupts

Interrupts are serviced on the Amiga through the use of interrupt *handlers* and *servers*. An interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. An interrupt server is one of possibly many system routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide a means of interrupt sharing. This concept is useful for general-purpose interrupts such as vertical blanking.

At system start, Exec designates certain interrupts as handlers and others as server chains. The **PORTS**, **COPER**, **VERTB**, **EXTER**, and **NMI** interrupts are initialized as server chains. Therefore, each of these may execute multiple interrupt routines per each interrupt. All other interrupts are designated as handlers and are always used exclusively.

INTERRUPT DATA STRUCTURE

Interrupt handlers and servers are defined by the Exec **Interrupt** structure. This structure specifies an interrupt routine entry point and data pointer. The C definition of this structure is as follows:

```
struct Interrupt
{
    struct Node is_Node;
    APTR        is_Data;
    VOID        (*is_Code)();
};
```

Once this structure has been properly initialized, it can be used for either a handler or a server.

ENVIRONMENT

Interrupts execute in an environment different from that of tasks. All interrupts execute in *supervisor mode* and utilize the single *system stack*. This stack is large enough to handle extreme cases of nested interrupts (of higher priorities). Interrupt processing has no effect on task stack usage.

All interrupt processing code, both handlers and servers, is invoked as assembly code subroutines. Normal assembly code register conventions dictate that the D0, D1, A0 and A1 registers be free for scratch use. In the case of an interrupt handler, some of these registers also contain data that may be useful to the handler code. See the section on handlers below.

Because interrupt processing executes outside the context of most system activities, certain data structures will not be self-consistent and must be considered off limits for all practical purposes. This happens because certain system operations are not atomic in nature and might be interrupted only after executing part of an important instruction sequence. For example, memory allocation and deallocation routines do not disable interrupts. This results in the possibility of interrupting a memory-related routine. In such a case, a memory linked list may be inconsistent during and interrupt. Therefore, interrupt routines must not use any memory allocation or deallocation functions.

In addition, interrupts may not call any system function which might allocate memory, wait, manipulate unprotected lists, or modify **ExecBase->ThisTask** data (for example **Forbid()**, **Permit()**, and **mathieee** libraries). In practice, this means that very few system calls may be used within interrupt code. The following functions may generally be used safely within interrupts:

```
Alert(), Disable(), Enable(), Signal(), Cause(),
GetMsg(), PutMsg(), ReplyMsg(), FindPort(), FindTask()
```

and if you are manipulating your *own* **List** structures while in an interrupt:

```
AddHead(), AddTail(), RemHead(), RemTail(), FindName()
```

In addition, certain devices (notably the timer device) specifically allow limited use of **SendIO()** and **BeginIO()** within interrupts.

INTERRUPT HANDLERS

As described above, an interrupt handler is a system routine that exclusively handles all processing related to a particular 4703 interrupt. There can only be one handler per 4703 interrupt. Every interrupt handler consists of an **Interrupt** structure (as defined above) and a single assembly code routine. Optionally, a data structure pointer may also be provided. This is particularly useful for ROM-resident interrupt code.

An interrupt handler is passed control as if it were a subroutine of Exec. Once the handler has finished its business, it must return to Exec by executing an `RTS` (return from subroutine) instruction rather than an `RTE` (return from exception) instruction. Interrupt handlers should be kept very short to minimize service-time overhead and thus minimize the possibilities of interrupt overruns. As described above, an interrupt handler has the normal scratch registers at its disposal. In addition, A5 and A6 are free for use. These registers are saved by Exec as part of the interrupt initiation cycle.

For the sake of efficiency, Exec passes certain register parameters to the handler (see the list below). These register values may be utilized to trim a few microseconds off the execution time of a handler. All of the following registers (D0/D1/A0/A1/A5/A6) may be used as scratch registers by an interrupt handler, and need not be restored prior to returning.

> *Don't Make Assumptions About Registers.* Interrupt servers have different register usage rules (see the ''Interrupt Servers'' section).

Interrupt Handler Register Usage

Here are the register conventions for interrupt handlers.

D0 Contains no valid information.

D1 Contains the 4703 **INTENAR** and **INTREQR** registers values AND'ed together. This results in an indication of which interrupts are enabled *and* active.

A0 Points to the base address of the Amiga custom chips. This information is useful for performing indexed instruction access to the chip registers.

A1 Points to the data area specified by the **is_Data** field of the **Interrupt** structure. Because this pointer is always fetched (regardless of whether you use it), it is to your advantage to make some use of it.

A5 Is used as a vector to your interrupt code.

A6 Points to the Exec library base (**SysBase**). You may use this register to call Exec functions or set it up as a base register to access your own library or device.

Interrupt handlers are established by passing the Exec function **SetIntVector()**, your initialized **Interrupt** structure, and the 4703 interrupt bit number of interest. The parameters for this function are as follows:

```
SetIntVector(ULONG intNumber, struct Interrupt *interrupt)
```

The first argument is the bit number for which this interrupt server is to respond (example **INTB_VERTB**). The possible bits for interrupts are defined in *<hardware/intbits.h>*. The second argument is the address of an interrupt server node as described earlier in this chapter. Keep in mind that certain interrupts are established as server chains and should not be accessed as handlers.

The following example demonstrates initialization and installation of an assembler interrupt handler. See the "Resources" chapter for more information on allocating resources, and the "Serial Device" chapter in the *Amiga ROM Kernel Reference Manual: Devices* for the more common method of serial communications.

```
;/* rbf.c - Execute me to compile me with SAS C 5.10
LC -d0 -b1 -cfistq -v -y -j73 rbf.c
Blink FROM LIB:c.o,rbf.o,rbfhandler.o TO rbf LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

** rbf.c - serial receive buffer full interrupt handler example.
** Must be linked with assembler handler rbfhandler.o
**
** To receive characters, this example requires ASCII serial input
** at your Amiga's current serial hardware baud rate (ie. 9600 after
** reboot, else last baud rate used)
*/

#include <exec/execbase.h>
#include <exec/memory.h>
#include <exec/interrupts.h>
#include <resources/misc.h>
#include <hardware/custom.h>
#include <hardware/intbits.h>
#include <dos/dos.h>

#include <clib/exec_protos.h>
#include <clib/misc_protos.h>

#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void)   { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

#define BUFFERSIZE 256

extern void RBFHandler();   /* proto for asm interrupt handler */
void main(void);

struct MiscResource *MiscBase;
extern struct ExecBase *SysBase;
extern struct Custom far custom;    /* defined in amiga.lib */

static UBYTE *allocname = "rbf-example";

struct RBFData {
    struct Task *rd_Task;
    ULONG rd_Signal;
    ULONG rd_BufferCount;
    UBYTE rd_CharBuffer[BUFFERSIZE + 2];
    UBYTE rd_FlagBuffer[BUFFERSIZE + 2];
    UBYTE rd_Name[32];
};

void main(void)
{
    struct RBFData *rbfdata;
    UBYTE *currentuser;
    BYTE signr;
    struct Device *serdevice;
    struct Interrupt *rbfint, *priorint;
    BOOL priorenable;
    ULONG signal;

    if (MiscBase = OpenResource("misc.resource"))
    {
        currentuser = AllocMiscResource(MR_SERIALPORT, allocname);       /* Allocate the serial */
        if (currentuser)                                                  /* port registers.     */
        {
            printf("serial hardware allocated by %s. Trying to remove it\n",
                   currentuser);                                         /* Hey! someone got it! */
```

```
        Forbid();
        if (serdevice = (struct Device *)FindName(&SysBase->DeviceList, currentuser))
            RemDevice(serdevice);
        Permit();

        currentuser = AllocMiscResource(MR_SERIALPORT, allocname);          /* and try again */
    }
    if (currentuser == NULL)
    {                                                                        /* Get the serial */
        currentuser = AllocMiscResource(MR_SERIALBITS, allocname);          /* control bits.  */
        if (currentuser)
        {
            printf("serial control allocated by %s\n", currentuser);           /* Give up. */
            FreeMiscResource(MR_SERIALPORT);
        }
        else
        {                                                                    /* Got them both. */
            printf("serial hardware allocated\n");
            if ((signr = AllocSignal(-1)) != -1)            /* Allocate a signal bit for the   */
            {                                                /* interrupt handler to signal us. */
                if (rbfint = AllocMem(sizeof(struct Interrupt), MEMF_PUBLIC|MEMF_CLEAR))
                {
                    if (rbfdata = AllocMem(sizeof(struct RBFData), MEMF_PUBLIC|MEMF_CLEAR))
                    {
                        rbfdata->rd_Task = FindTask(NULL);       /* Init rfbdata structure. */
                        rbfdata->rd_Signal = 1L << signr;

                        rbfint->is_Node.ln_Type = NT_INTERRUPT;      /* Init interrupt node. */
                        strcpy(rbfdata->rd_Name, allocname);
                        rbfint->is_Node.ln_Name = rbfdata->rd_Name;
                        rbfint->is_Data = (APTR)rbfdata;
                        rbfint->is_Code = RBFHandler;
                                                                   /* Save state of RBF and */
                        priorenable = custom.intenar & INTF_RBF ? TRUE : FALSE; /* interrupt */
                        custom.intena = INTF_RBF;                              /* disable it. */
                        priorint = SetIntVector(INTB_RBF, rbfint);

                        if (priorint) printf("replaced the %s RBF interrupt handler\n",
                                             priorint->is_Node.ln_Name);
                        printf("enabling RBF interrupt\n");
                        custom.intena = INTF_SETCLR | INTF_RBF;

                        printf("waiting for buffer to fill up. Use CTRL-C to break\n");
                        signal = Wait(1L << signr | SIGBREAKF_CTRL_C);

                        if (signal & SIGBREAKF_CTRL_C) printf(">break<\n");
                        printf("Character buffer contains:\n%s\n", rbfdata->rd_CharBuffer);

                        custom.intena = INTF_RBF;             /* Restore previous handler. */
                        SetIntVector(INTB_RBF, priorint);
                                                            /* Enable it if it was enabled */
                        if (priorenable) custom.intena = INTF_SETCLR|INTF_RBF;     /* before. */

                        FreeMem(rbfdata, sizeof(struct RBFData));
                    }
                    else  printf("can't allocate memory for rbf data\n");
                    FreeMem(rbfint, sizeof(struct Interrupt));
                }
                else printf("can't allocate memory for interrupt structure\n");
                FreeSignal(signr);
            }
            else printf("can't allocate signal\n");

            FreeMiscResource(MR_SERIALBITS);   /* release serial hardware */
            FreeMiscResource(MR_SERIALPORT);
        }
    }
  } /* There is no 'CloseResource()' function */
}
```

The assembler interrupt handler code, **RBFHandler**, reads the complete word of serial input data from the serial hardware and then separates the character and flag bytes into separate buffers. When the buffers are full, the handler signals the main process causing main to print the character buffer contents, remove the handler, and exit.

NOTE. The data structure containing the signal to use, task address pointer, and buffers is allocated and initialized in **main()**, and passed to the handler (shown below) via the **is_Data** pointer of the **Interrupt** structure.

```
* rbfhandler.asm. Example interrupt handler for rbf.
*
* Assembled with Howesoft Adapt 680x0 Macro Assembler Rel. 1.0
* hx68 from: rbfhandler.asm to rbfhandler.o INCDIR include:
* blink from lib:c.o rbf.o rbfhandler.o to rbf lib lib:lc.lib lib:amiga.lib
*
    INCLUDE "exec/types.i"
    INCLUDE "hardware/custom.i"
    INCLUDE "hardware/intbits.i"

        XDEF    _RBFHandler

JSRLIB MACRO
       XREF _LVO\1
       JSR  _LVO\1(A6)
       ENDM

BUFLEN    EQU    256

       STRUCTURE RBFDATA,0
        APTR   rd_task
        ULONG  rd_signal
        UWORD  rd_buffercount
        STRUCT rd_charbuffer,BUFLEN+2
        STRUCT rd_flagbuffer,BUFLEN+2
        STRUCT rd_name,32
        LABEL RBFDATA_SIZEOF

* Entered with:
*  D0 == scratch
*  D1 == INTENAT & INTREQR (scratch)
*  A0 == custom chips (scratch)
*  A1 == is_Data which is RBFDATA structure (scratch)
*  A5 == vector to our code (scratch)
*  A6 == pointer to ExecBase (scratch)
*
* Note - This simple handler just receives one buffer full of serial
* input data, signals main, then ignores all subsequent serial data.
*
    section code

_RBFHandler:                            ;entry to our interrupt handler

        MOVE.W  serdatr(A0),D1          ;get the input word (flags and char)

        MOVE.W  rd_buffercount(A1),D0   ;get our buffer index
        CMPI.W  #BUFLEN,D0              ;no more room in our buffer ?
        BEQ.S   ExitHandler             ;yes - just exit (ignore new char)
        LEA.L   rd_charbuffer(A1),A5    ;else get our character buffer address
        MOVE.B  D1,0(A5,D0.W)           ;store character in our character buffer
        LEA.L   rd_flagbuffer(A1),A5    ;get our flag buffer address
        LSR.W   #8,d1                   ;shift flags down
        MOVE.B  D1,0(A5,D0.W)           ;store flags in flagbuffer

        ADDQ.W  #1,D0                   ;increment our buffer index
        MOVE.W  D0,rd_buffercount(A1)   ;   and replace it
        CMPI.W  #BUFLEN,D0              ;did our buffer just become full ?
        BNE.S   ExitHandler             ;no - we can exit
        MOVE.L  A0,-(SP)                ;yes - save custom
        MOVE.L  rd_signal(A1),D0        ;get signal allocated in main()
        MOVE.L  rd_task(A1),A1          ;and pointer to main task
        JSRLIB  Signal                  ;tell main we are full
        MOVE.L  (SP)+,A0                ;restore custom
                                        ;Note: system call trashed D0-D1/A0-A1
ExitHandler:
        MOVE.W  #INTF_RBF,intreq(A0)    ;clear the interrupt
        RTS                             ;return to exec

        END
```

INTERRUPT SERVERS

As mentioned above, an interrupt server is one of possibly many system interrupt routines that are invoked as the result of a single 4703 interrupt. Interrupt servers provide an essential mechanism for interrupt sharing.

Interrupt servers must be used for **PORTS**, **COPER**, **VERTB**, **EXTER**, or **NMI** interrupts. For these interrupts, all servers are linked together in a chain. Every server in the chain will be called in turn as long as the previous server returned with the processor's **Z** (zero) flag set. If you determine that an interrupt was specifically for your server, you should return with the processor's **Z** flag cleared (non-zero condition) so that the remaining servers on the chain will be skipped.

> *Use The Z Flag.* **VERTB** (vertical blank) servers should *always* return with the **Z** (zero) flag set. The processor **Z** flag is used rather than the normal function convention of returning a result in D0 because it may be tested more quickly by Exec upon the server's return.

The easiest way to set the condition code register is to do an immediate move to the D0 register as follows:

```
SetZflag_Calls_Next:
        MOVEQ   #0,D0
        RTS

ClrZflag_Ends_Chain:
        MOVEQ   #1,D0
        RTS
```

The same Exec **Interrupt** structure used for handlers is also used for servers. Also, like interrupt handlers, servers must terminate their code with an `RTS` instruction.

Interrupt servers are called in priority order. The priority of a server is specified in its **is_Node.ln_Pri** field. Higher-priority servers are called earlier than lower-priority servers. Adding and removing interrupt servers from a particular chain is accomplished with the Exec **AddIntServer()** and **RemIntServer()** functions. These functions require you to specify both the 4703 interrupt number and a properly initialized **Interrupt** structure.

Servers have different register values passed than handlers do. A server cannot count on the D0, D1, A0, or A6 registers containing any useful information. However, the highest priority system vertical blank server currently expects to receive a pointer to the custom chips A0. Therefore, if you install a vertical blank server at priority 10 or greater, you must place custom ($DFF000) in A0 before exiting. Other than that, a server is free to use D0-D1 and A0-A1/A5-A6 as scratch.

Interrupt Server Register Usage

D0 Scratch.

D1 Scratch.

A0 Scratch except in certain cases (see note above).

A1 Points to the data area specified by the **is_Data** field of the **Interrupt** structure. Because this pointer is always fetched (regardless of whether you use it), it is to your advantage to make some use of it (scratch).

A5 Points to your interrupt code (scratch).

A6 Scratch.

In a server chain, the interrupt is cleared automatically by the system. Having a server clear its interrupt is not recommended and not necessary (clearing could cause the loss of an interrupt on **PORTS** or **EXTER**).

Here is an example of a program to install and remove a low-priority vertical blank interrupt server:

```
;/* vertb.c - Execute me to compile me with SAS C 5.10
LC -d0 -b1 -cfistq -v -y -j73 vertb.c
Blink FROM LIB:c.o,vertb.o,vertbserver.o TO vertb LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ; */
/* vertb.c - Vertical blank interrupt server example.  Must be linked with vertbserver.o. */

#include <exec/memory.h>
#include <exec/interrupts.h>
#include <dos/dos.h>
#include <hardware/custom.h>
#include <hardware/intbits.h>
#include <clib/exec_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)    { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

extern void VertBServer();  /* proto for asm interrupt server */

void main(void)
{
    struct Interrupt *vbint;
    ULONG counter = 0;
    ULONG endcount;
                                                                      /* Allocate memory for  */
    if (vbint = AllocMem(sizeof(struct Interrupt), MEMF_PUBLIC|MEMF_CLEAR))   /* interrupt node. */
    {
        vbint->is_Node.ln_Type = NT_INTERRUPT;         /* Initialize the node. */
        vbint->is_Node.ln_Pri = -60;
        vbint->is_Node.ln_Name = "VertB-Example";
        vbint->is_Data = (APTR)&counter;
        vbint->is_Code = VertBServer;


        AddIntServer(INTB_VERTB, vbint); /* Kick this interrupt server to life. */

        printf("VBlank server will increment a counter every frame.\n");
        printf("counter started at zero, CTRL-C to remove server\n");

        Wait(SIGBREAKF_CTRL_C);
        endcount = counter;
        printf("%ld vertical blanks occurred\nRemoving server\n", endcount);

        RemIntServer(INTB_VERTB, vbint);
        FreeMem(vbint, sizeof(struct Interrupt));
    }
    else printf("Can't allocate memory for interrupt node\n");
}
```

This is the assembler **VertBServer** installed by the C example:

```
* vertbserver.asm. Example simple interrupt server for vertical blank
*
* Assembled with Howesoft Adapt 680x0 Macro Assembler Rel. 1.0
* hx68 from: vertbserver.asm to vertbserver.o INCDIR include:
* blink from lib:c.o vertb.o vertbserver.o to vertb lib lib:lc.lib lib:amiga.lib
*
    INCLUDE "exec/types.i"
    INCLUDE "hardware/custom.i"
    INCLUDE "hardware/intbits.i"

        XDEF    _VertBServer

* Entered with:        A0 == scratch (execpt for highest pri vertb server)
*  D0 == scratch       A1 == is_Data
*  D1 == scratch       A5 == vector to interrupt code (scratch)
*                      A6 == scratch
*
    section code

_VertBServer:
        ADDI.L  #1,(a1)          ; increments counter is_Data points to
        MOVEQ.L #0,d0            ; set Z flag to continue to process other vb-servers
        RTS                      ;return to exec
        END
```

Software Interrupts

Exec provides a means of generating *software interrupts*. Software interrupts execute at a priority higher than that of tasks but lower than that of hardware interrupts, so they are often used to defer hardware interrupt processing to a lower priority. Software interrupts use the same **Interrupt** data structure as hardware interrupts. As described above, this structure contains pointers to both interrupt code and data, and should be initialized as node type NT_INTERRUPT (not NT_SOFTINT which is an internal Exec flag).

A software interrupt is usually activated with the **Cause()** function. If this function is called from a task, the task will be interrupted and the software interrupt will occur. If it is called from a hardware interrupt, the software interrupt will not be processed until the system exits from its last hardware interrupt. If a software interrupt occurs from within another software interrupt, it is not processed until the current one is completed. However, individual software interrupts do not nest, and will not be caused if already running as a software interrupt.

> *Don't Trash A6!* Software interrupts execute in an environment almost identical to that of hardware interrupts, and the same restrictions on allowable system function calls (as described earlier) apply to both. Note however that, unlike other interrupts, software interrupts *must* preserve A6.

Software interrupts are prioritized. Unlike interrupt servers, software interrupts have only five allowable priority levels: -32, -16, 0, +16, and +32. The priority should be put into the **ln_Pri** field prior to calling **Cause()**.

Software interrupts can also be generated by message arrival at a PA_SOFTINT message port. The applications of this technique are limited since it is not permissible, with most devices, to send IO requests from within interrupt code. However, the timer.device does allow such interactions, so a self-perpetuating PA_SOFTINT timer port can provide an application with quite consistent timing under varying multitasking loads. The following example demonstrates use of a software interrupt and a PA_SOFTINT port. See the ''Exec Messages and Ports'' chapter for more information about messages and ports.

```
;/* timersoftint.c - Execute me to compile me with SAS C 5.10
LC -b1 -d0 -cfistq -v -y -j73 timersoftint.c
Blink FROM LIB:c.o,timersoftint.o TO timersoftint LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ; */
/* timersoftint.c - Timer device software interrupt message port example. */

#include <exec/memory.h>
#include <exec/interrupts.h>
#include <devices/timer.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/alib_protos.h>
#include <stdio.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }  /* really */
#endif

#define MICRO_DELAY 1000
#define OFF     0
#define ON      1
#define STOPPED 2

struct TSIData {
    ULONG tsi_Counter;
    ULONG tsi_Flag;
    struct MsgPort *tsi_Port;
};

struct TSIData *tsidata;

void tsoftcode(void);    /* Prototype for our software interrupt code */

void main(void)
{
    struct MsgPort *port;
    struct Interrupt *softint;
    struct timerequest *tr;

    ULONG endcount;

    /* Allocate message port, data & interrupt structures. Don't use CreatePort() */
    /* or CreateMsgPort() since they allocate a signal (don't need that) for a    */
    /* PA_SIGNAL type port. We need PA_SOFTINT.                                    */
    if (tsidata = AllocMem(sizeof(struct TSIData), MEMF_PUBLIC|MEMF_CLEAR))
    {
        if(port = AllocMem(sizeof(struct MsgPort), MEMF_PUBLIC|MEMF_CLEAR))
        {
            NewList(&(port->mp_MsgList));                         /* Initialize message list */
            if (softint = AllocMem(sizeof(struct Interrupt), MEMF_PUBLIC|MEMF_CLEAR))
            {
                /* Set up the (software)interrupt structure. Note that this task runs at  */
                /* priority 0. Software interrupts may only be priority -32, -16, 0, +16, */
                /* +32. Also not that the correct node type for a software interrupt is   */
                /* NT_INTERRUPT. (NT_SOFTINT is an internal Exec flag). This is the same  */
                /* setup as that for a software interrupt which you Cause(). If our       */
                /* interrupt code was in assembler, you could initialize is_Data here to  */
                /* contain a pointer to shared data structures. An assembler software     */
                /* interrupt routine would receive the is_Data in A1.                     */

                softint->is_Code = tsoftcode;    /* The software interrupt routine */
                softint->is_Data = tsidata;
                softint->is_Node.ln_Pri = 0;

                port->mp_Node.ln_Type = NT_MSGPORT;       /* Set up the PA_SOFTINT message port  */
                port->mp_Flags = PA_SOFTINT;              /* (no need to make this port public). */
                port->mp_SigTask = (struct Task *)softint;     /* pointer to interrupt structure */

                /* Allocate timerequest */
                if (tr = (struct timerequest *) CreateExtIO(port, sizeof(struct timerequest)))
                {
                    /* Open timer.device. NULL is success. */
                    if (!(OpenDevice("timer.device", UNIT_MICROHZ, (struct IORequest *)tr, 0)))
                    {
```

```
                    tsidata->tsi_Flag = ON;         /* Init data structure to share globally. */
                    tsidata->tsi_Port = port;

                    /* Send of the first timerequest to start. IMPORTANT: Do NOT    */
                    /* BeginIO() to any device other than audio or timer from       */
                    /* within a software or hardware interrupt. The BeginIO() code */
                    /* may allocate memory, wait or perform other functions which  */
                    /* are illegal or dangerous during interrupts.                  */
                    printf("starting softint. CTRL-C to break...\n");


                    tr->tr_node.io_Command = TR_ADDREQUEST;     /* Initial iorequest to start */
                    tr->tr_time.tv_micro = MICRO_DELAY;         /* software interrupt.        */
                    BeginIO((struct IORequest *)tr);

                    Wait(SIGBREAKF_CTRL_C);
                    endcount = tsidata->tsi_Counter;
                    printf("timer softint counted %ld milliseconds.\n", endcount);

                    printf("Stopping timer...\n");
                    tsidata->tsi_Flag = OFF;

                    while (tsidata->tsi_Flag != STOPPED) Delay(10);

                    CloseDevice((struct IORequest *)tr);
                }
                else printf("couldn't open timer.device\n");
                DeleteExtIO(tr);
            }
            else printf("couldn't create timerequest\n");
            FreeMem(softint, sizeof(struct Interrupt));
        }
        FreeMem(port, sizeof(struct MsgPort));
    }
    FreeMem(tsidata, sizeof(struct TSIData));
  }
}


void tsoftcode(void)
{
    struct timerequest *tr;

    /* Remove the message from the port. */
    tr = (struct timerequest *)GetMsg(tsidata->tsi_Port);

    /* Keep on going if main() hasn't set flag to OFF. */
    if ((tr) && (tsidata->tsi_Flag == ON))
    {
        /* increment counter and re-send timerequest--IMPORTANT: This         */
        /* self-perpetuating technique of calling BeginIO() during a software */
        /* interrupt may only be used with the audio and timer device.        */
        tsidata->tsi_Counter++;
        tr->tr_node.io_Command = TR_ADDREQUEST;
        tr->tr_time.tv_micro = MICRO_DELAY;
        BeginIO((struct IORequest *)tr);
    }
    /* Tell main() we're out of here. */
    else tsidata->tsi_Flag = STOPPED;
}
```

Disabling Interrupts

As mentioned in the "Exec Tasks" chapter, it is sometimes necessary to disable interrupts when examining or modifying certain shared system data structures. However, for proper system operation, interrupts should never be disabled unless absolutely necessary, and *never for more than 250 microseconds*. Interrupt disabling is controlled with the **Disable()** and **Enable()** functions. Although assembler **DISABLE** and **ENABLE** macros are provided, we strongly suggest that you use the system functions rather than the macros for upwards compatibility and smaller code size.

In some system code, there are nested disabled sections. Such code requires that interrupts be disabled with the first **Disable()** and not re-enabled until the *last* **Enable()**. The system **Enable()** and **Disable()** functions are designed to permit this sort of nesting.

Disable() increments a counter to track how many levels of disable have been issued. Only 126 levels of nesting are permitted. **Enable()** decrements the counter, and reenables interrupts when the last disable level has been exited.

Function Reference

The following chart gives a brief description of the Exec functions that control interrupts. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details about each call.

Table 26-2: Exec Interrupt Functions

Interrupt Function	Description
AddIntServer()	Add an interrupt server to a system server chain.
Cause()	Cause a software interrupt.
Disable()	Disable interrupt processing.
Enable()	Restart system interrupt processing.
RemIntServer()	Remove an interrupt server from a system server chain.
SetIntVector()	Set a new handler for a system interrupt vector.

Chapter 27
GRAPHICS PRIMITIVES

This chapter describes the basic graphics functions available to Amiga programmers. It covers the graphics support structures, display routines and drawing routines. Many of the operations described in this section are also performed by the Intuition software. See the Intuition chapters for more information.

The Amiga supports several basic types of graphics routines: display routines, drawing routines, sprites and animation. These routines are very versatile and allow you to define any combination of drawing and display areas you may wish to use.

The first section of this chapter defines the display routines. These routines show you how to form and manipulate a display, including the following aspects of display use:

- How to query the graphics system to find out what type of video monitor is attached and which graphics modes can be displayed on it.
- How to identify the memory area that you wish to have displayed.
- How to position the display area window to show only a certain portion of a larger drawing area.
- How to split the screen into as many vertically stacked slices as you wish.
- How to determine which horizontal and vertical resolution modes to use.
- How to determine the current correct number of pixels across and lines down for a particular section of the display.
- How to specify how many color choices per pixel are to be available in a specific section of the display.

The later sections of the chapter explain all of the available modes of drawing supported by the system software, including how to do the following:

- Reserve memory space for use by the drawing routines.
- Define the colors that can be drawn into a drawing area.
- Define the colors of the drawing pens (foreground pen, background pen for patterns, and outline pen for area-fill outlines).
- Define the pen position in the drawing area.
- Drawing primitives; lines, rectangles, circles and ellipses.
- Define vertex points for area-filling, and specify the area-fill color and pattern.
- Define a pattern for patterned line drawing.
- Change drawing modes.
- Read or write individual pixels in a drawing area.
- Copy rectangular blocks of drawing area data from one drawing area to another.
- Use a template (predefined shape) to draw an object into a drawing area.

COMPONENTS OF A DISPLAY

In producing a display, you are concerned with two primary components: sprites and the playfield. Sprites are the easily movable parts of the display. The playfield is the static part of the display and forms a backdrop against which the sprites can move and with which the sprites can interact.

This chapter covers the creation of the background. Sprites are described in the ''Graphics Sprites, Bobs and Animation'' chapter.

INTRODUCTION TO RASTER DISPLAYS

There are three major television standards in common use around the world: NTSC, PAL, and SECAM. NTSC is used primarily in the United States and Japan; PAL and SECAM are used primarily in Europe. The Amiga currently supports both NTSC and PAL. The major differences between the two systems are refresh frequency and the number of scan lines produced. Where necessary, the differences will be described and any special considerations will be mentioned.

The Amiga produces its video displays on standard television or video monitors by using raster display techniques. The picture you see on the video display screen is made up of a series of horizontal video lines stacked one on top of another, as illustrated in the following figure. Each line represents one sweep of an electronic video beam, which ''paints'' the picture as it moves along. The beam sweeps from left to right, producing the full screen one line at a time. After producing the full screen, the beam returns to the top of the display screen.

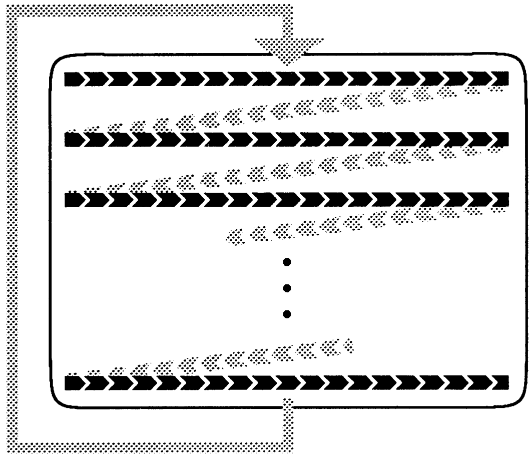


Figure 27-1: How the Video Display Picture Is Produced

The diagonal lines in the figure show how the video beam returns to the start of each horizontal line.

Effect of Display Overscan on the Viewing Area

To assure that the picture entirely fills the monitor (or television) screen, the manufacturer of the video display device usually creates a deliberate *overscan*. That is, the video beam is swept across an area that is larger than the viewable region of the monitor.

The video beam actually covers 262 vertical lines (312 for PAL). The user, however, sees only the portion of the picture that is within the center region of the display, typically surrounded by a border as illustrated in the figure below. The center region is nominally about 200 lines high on an NTSC machine (256 lines for PAL). Overscan also limits the amount of video data that can appear on each display line. The width of the center region is nominally, about 320 pixels for both PAL and NTSC.

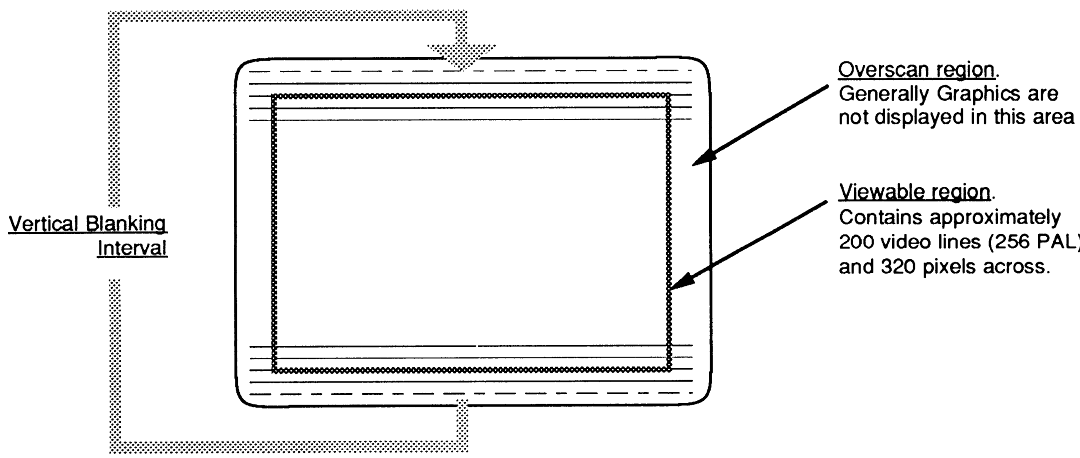


Figure 27-2: Display Overscan Restricts Usable Picture Area

The flexibility of the Amiga graphics subsystem allows the overscan region, which normally forms the border of the display, to be used for application graphics instead. So the nominal dimensions given above can be enlarged.

The time during which the video beam is below the bottom line of the viewable region and above the first line is called the *vertical blanking interval.* The recommended minimum to allow for this interval is 21 lines for NTSC (29 lines for PAL). So, for applications that take full advantage of the overscan area, a maximum of 241 usable lines in NTSC (283 in PAL) can be achieved. The display resolution can also be changed by changing the Amiga display mode as discussed in the sections below.

Color Information for the Video Lines

The hardware reads the system display memory to obtain the color information for each line. As the video display beam sweeps across the screen producing the display line, it changes color, producing the images you have defined. On the current generation of Amiga hardware, there are 4,096 possible colors.

INTERLACED AND NON-INTERLACED MODES

In producing the complete display (262 lines in NTSC, 312 in PAL), the video display device produces the top line, then the next lower line, then the next, until it reaches the bottom of the screen. When it reaches the bottom, it returns to the top to start a new scan of the screen. Each complete set of lines is called a *display field.* It takes about 1/60th of a second to produce a complete NTSC display field (1/50th of a second for PAL).

The Amiga has two vertical display modes: *interlaced* and *non-interlaced.* In non-interlaced mode, the video display produces the same picture for each successive display field. A non-interlaced NTSC display normally has about 200 lines in the viewable area (up to a maximum of 241 lines with overscan) while a PAL display will normally show 256 lines (up to a maximum of 283 with overscan).

With interlaced mode, the amount of information in the viewable area can be doubled. On an NTSC display this amounts to 400 lines (482 with overscan), while on a PAL display it amounts to 512 lines (566 with overscan).

For interlaced mode, the video beam scans the screen at the same rate (1/60th of a second per complete NTSC video display field); however, it takes two display fields to form a complete video display picture and twice as much display memory to store line data. During the first of each pair of display fields, the system hardware shows the odd-numbered lines of an interlaced display (1, 3, 5, and so on). During the second display field, it shows the even-numbered lines (2, 4, 6 and so on). The second field is positioned slightly lower so that the lines in the second field are ''interlaced'' with those of the first field, giving the higher vertical resolution of this mode.

Data as Displayed	Data in Memory
Odd field - Line 1	Line 1
Even field - Line 1	Line 2
Odd field - Line 2	Line 3
Even field - Line 2	Line 4
⋮	⋮
Odd field - Line 200	Line 399
Even field - Line 200	Line 400

Figure 27-3: Interlaced Mode — Display Fields and Data in Memory

The following figure shows a display formed as display lines 1, 2, 3, 4, ... 400. The 400-line interlaced display uses the same physical display area as a 200-line non-interlaced display.

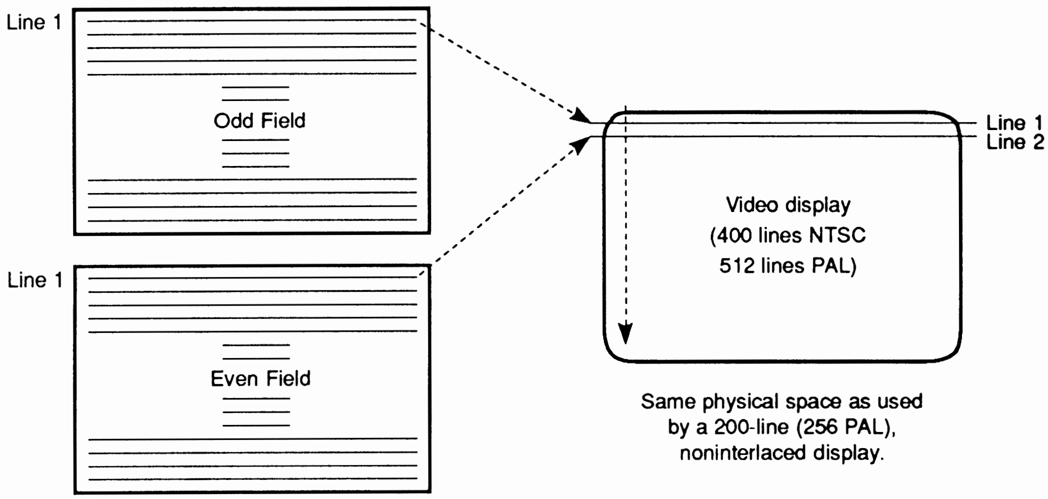


Figure 27-4: Interlaced Mode Doubles Vertical Resolution

During an interlaced display, it appears that both display fields are present on the screen at the same time and form one complete picture. However, interlaced displays will appear to flicker if adjacent (odd and even) scan lines have contrasting brightness. Choosing appropriate colors for your display will reduce this flicker considerably. This phenomenon can also be reduced by using a long-persistence monitor, or alleviated completely with a hardware de-interlacer.

LOW, HIGH AND SUPER-HIGH RESOLUTION MODES

The Amiga also has three horizontal display modes: low-resolution (or *Lores*), high-resolution (*Hires*) and super-high-resolution (*SuperHires*).

Normally, these three horizontal display modes have a width of 320 for Lores, 640 for Hires or 1280 for SuperHires on both PAL and NTSC machines. However, by taking full advantage of the overscan region, it is possible to create dispays up to 362 pixels wide in Lores mode, 724 pixels wide in Hires or 1448 pixels wide in SuperHires. Usually, however, you should use the standard values (320, 640 or 1280) for most applications.

In general, the number of colors available in each display mode decreases as the resolution increases. The Amiga has two special display modes that can be used to increase the number of colors available. HAM is *Hold-And-Modify* mode, EHB is *Extra-Half-Brite* mode.

Hold-And-Modify (HAM) allows you to display the entire palette of 4,096 colors on-screen at once with certain restrictions, explained later.

Extra-Half-Brite allows for 64 colors on-screen at once; 32 colors plus 32 additional colors that are half the intensity of the first 32. For example, if color 1 is defined as 0xFFF (white), then color 33 is 0x777 (grey).

Display Modes, Colors, and Requirements

The following chart lists all of the display modes that are available under Release 2 of the Amiga operating system, as well as those available under previous releases of the OS.

15 kHz Amiga Display Modes	**Default Resolution NTSC**	**PAL**	**Maximum Colors**	**Supports HAM/EHB**
Lores	320x200	320x256	32 of 4096	Yes
Lores-Interlaced	320x400	320x512	32 of 4096	Yes
Hires	640x200	640x256	16 of 4096	No
Hires-Interlaced	640x400	640x512	16 of 4096	No
SuperHires*	1280x200	1280x256	4 out of 64	No
SuperHires-Interlaced*	1280x400	1280x512	4 out of 64	No

*Requires both Release 2 and ECS.

31 kHz Amiga Display Modes*	**Default Resolution**	**Maximum Colors**	**Supports HAM/EHB**
VGA-ExtraLores	160x480	32 out of 4096	Yes
VGA-ExtraLores-Interlace	160x960	32 out of 4096	Yes
VGA-Lores	320x480	16 out of 4096	No
VGA-Lores-Interlace	320x960	16 out of 4096	No
Productivity	640x480	4 out of 64	No
Productivity-Interlace	640x960	4 out of 64	No

*31 kHz modes require Release 2, ECS and either a bi-scan or multi-scan monitor.

A2024* Display Modes	**Default Resolution NTSC**	**PAL**	**Maximum Colors**
A2024-10Hz	1008x800	1008x1024	4 out of 4 grey levels
A2024-15Hz	1008x800	1008x1024	4 out of 4 grey levels

*A2024 modes require special hardware and either Release 2 or special software available from the monitor's manufacturer.

ABOUT ECS

ECS stands for *Enhanced Chip Set*, the latest version of the Amiga's custom chips that provides for improved graphics capabilities. Some of the special features of the Amiga's graphics sub-system such as the VGA, Productivity and SuperHires display modes require the ECS.

SuperHires (35 nanosecond) Pixel Resolutions

The enhanced version of the Denise chip can generate SuperHires pixels that are twice as fine as Hires pixels. It is convenient to refer to pixels here by their speed, rather than width, for reasons that will be explained below. They are approximately 35nS long, while Hires are 70nS, and Lores 140nS. In the absence of any other features, this can bring a new mode with nominal dimensions of 1280 x 200 (NTSC) or 1280 x 256 (PAL). This mode requires the ECS Agnus chip as well.

When Denise is generating these new fast pixels, simple bandwidth arithmetic indicates that at most two bitplanes can be supported. Also note that with two bitplanes, DMA bandwidth is saturated. The palette for SuperHires pixels is also restricted to 64 colors.

Productivity Mode

The enhanced version of the Denise chip can support monitor horizontal scan frequencies of 31KHz, twice the old 15.75KHz rate. This provides over 400 non-interlaced horizontal lines in a frame, but requires the use of a multiple scan rate, or multi-sync monitor.

This effect speeds up the video beam roughly by a factor of two, which has the side effect of doubling the width of a pixel emitted at a given speed. Thus, for a given Denise mode, pixels are twice as fat, and there are half as many on a given line.

The increased scan rate interacts with all of the Denise modes. So with both SuperHires (35nS) pixels and the double scan rate the display generated would be 640 pixels wide by more than 400 rows, non-interlaced, with up to four colors from a palette of 64. This combination is termed Productivity mode, and the default international height is 480 rows. This conforms, in a general way, to the VGA Mode 3 Standard 8514/A.

The support in Agnus is actually more flexible, and gives the ability to conform to special-purpose modes, such as displays synchronized to motion picture cameras.

Selectable PAL/NTSC

The Enhanced Chip Set can be set to NTSC or PAL modes under software control. Its initial default behavior is determined by a jumper or trace on the system motherboard. This has no bearing on Productivity mode and other programmable scan operations, but the new system software can support displays in either mode.

Determining Chip Versions

It is possible to ascertain whether the ECS chips are in the machine at run time by looking in the **ChipRevBits0** field of the **GfxBase** structure. If this field contains the flag for the chip you are interested in (as defined in the *<gfxbase.h>* include file), then that chip is present.

For example, if the C statement `(GfxBase->ChipRevBits0 & GFXF_HR_AGNUS)` evaluates to non-zero, then the machine contains the ECS version of the Agnus chip and has advanced features such as the ability to handle larger rasters. Older Agnus chips were capable of handling rasters up to 1,024 by 1,024 pixels. The ECS Agnus can handle rasters up to 16,384 by 16,384 pixels.

If `(GfxBase->ChipRevBits0 & GFXF_HR_DENISE)` is non-zero, then the ECS version of the Denise chip is present. Having both the ECS Agnus and ECS Denise present allows for the special SuperHires, VGA and Productivity display modes available in Release 2. For more information on ECS and the custom chips, refer to the *Amiga Hardware Reference Manual.*

FORMING AN IMAGE

To create an image, you write data (that is, you ''draw'') into a memory area in the computer. From this memory area, the system can retrieve the image for display. You tell the system exactly how the memory area is organized, so that the display is correctly produced. You use a block of memory words at sequentially increasing addresses to represent a rectangular region of data bits. The following figure shows the contents of three example memory words: bits are shown as blank rectangles, and 1 bits as filled-in rectangles.

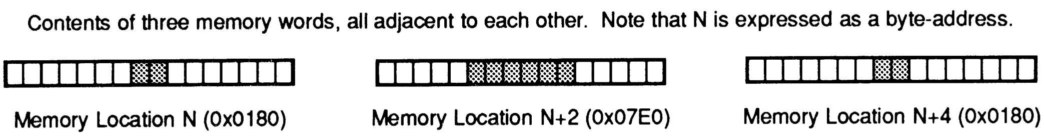


Figure 27-5: Sample Memory Words

The system software lets you define linear memory as rectangular regions, called *bitplanes*. The figure below shows how the system would organize three sequential words in memory into a rectangular bitplane with dimensions of 16 x 3 pixels.

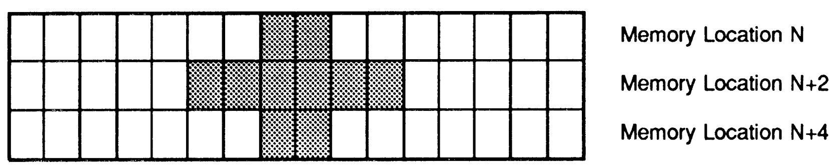


Figure 27-6: A Rectangular Bitplane Made from 3 Memory Words

The following figure shows how 4,000 words (8,000 bytes) of memory can be organized to provide enough bits to define a single bitplane of a full-screen, low-resolution video display (320 x 200).

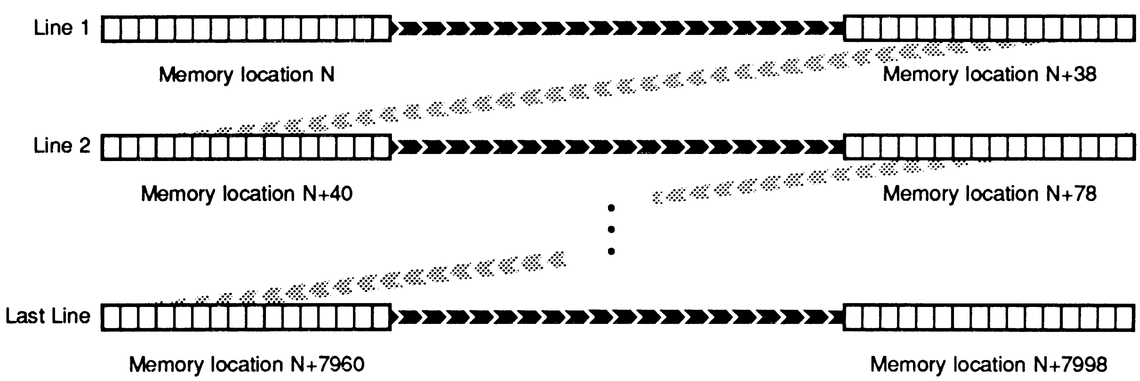


Figure 27-7: Bitplane for a Full-screen, Low-resolution Display

Each memory data word contains 16 data bits. The color of each pixel on a video display line is directly related to the value of one or more data bits in memory, as follows:

- If you create a display in which each pixel is related to only one data bit, you can select from only two possible colors, because each bit can have a value of only 0 or 1.
- If you use two bits per pixel, there is a choice of four different colors because there are four possible combinations of the values of 0 and 1 from each of the two bits.
- If you specify three, four, or five bits per pixel, you will have eight, sixteen, or thirty-two possible choices of a color for a pixel.
- If you use six bits per pixel, then depending on the video mode (EHB or HAM), you will have sixty-four or 4,096 possible choices for a pixel.

To create multicolored images, you must tell the system how many bits are to be used per pixel. The number of bits per pixel is the same as the *number of bitplanes* used to define the image.

As the video beam sweeps across the screen, the system retrieves one data bit from each bitplane. Each of the data bits is taken from a different bitplane, and one or more bitplanes are used to fully define the video display screen. For each pixel, data-bits in the same x,y position in each bitplane are combined by the system hardware to create a binary value. This value determines the color that appears on the video display for that pixel.

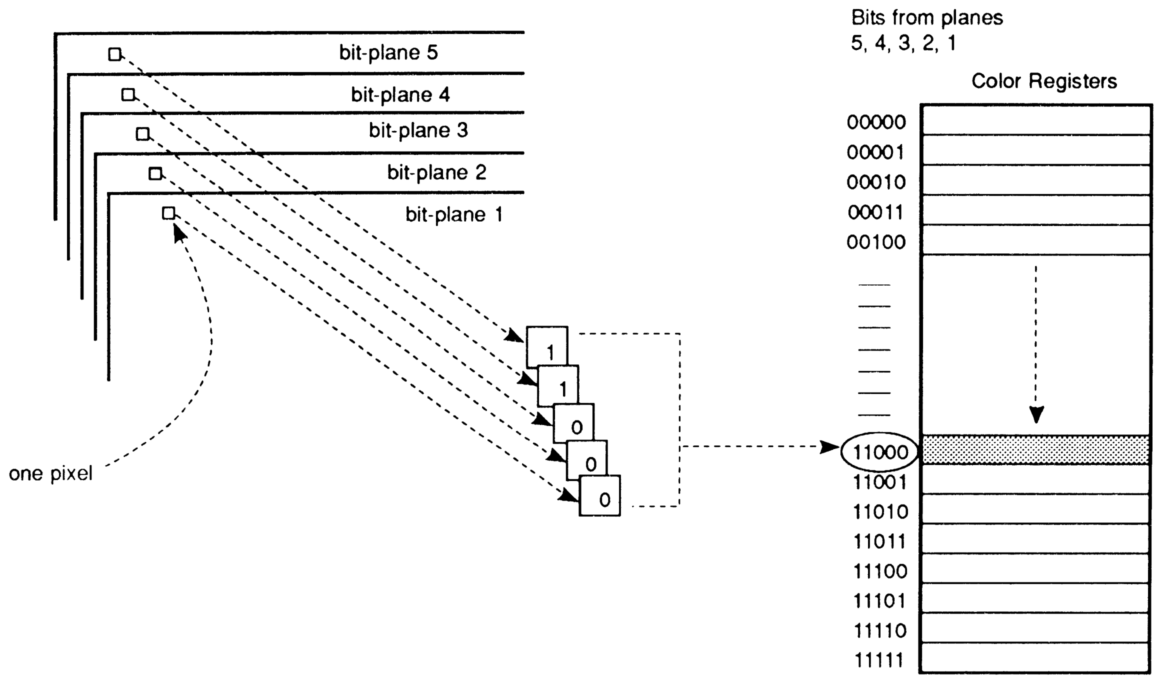


Figure 27-8: Bits from Each Bitplane Select Pixel Color

You will find more information showing how the data bits actually select the color of the displayed pixel in the section called "ViewPort Color Selection."

ROLE OF THE COPPER (COPROCESSOR)

The Amiga has a special-purpose coprocessor, called the *Copper*, that can control nearly the entire graphics system. The Copper can control register updates, reposition sprites, change the color palette, and update the blitter. The graphics and animation routines use the Copper to set up lists of instructions for handling displays, and advanced programmers can create their own custom Copper lists.

Display Routines and Structures

> *CAUTION:* This section describes the lowest-level graphics interface to the system hardware. If you use any of the routines and the data structures described in these sections, your program will essentially take over the entire display. In general, this is not compatible with Intuition's multiwindow operating environment since Intuition calls these low-level routines for you.

The descriptions of the display routines, as well as those of the drawing routines, occasionally use the same terminology as that in the Intuition chapters. These routines and data structures are the same ones that Intuition software uses to produce its displays.

The computer produces a display from a set of instructions you define. You organize the instructions as a set of parameters known as the **View** structure (see the *<graphics/view.h>* include file for more information). The following figure shows how the system interprets the contents of a **View** structure. This drawing shows a complete display composed of two different component parts, which could (for example) be a low-resolution, multicolored part and a high-resolution, two-colored part.

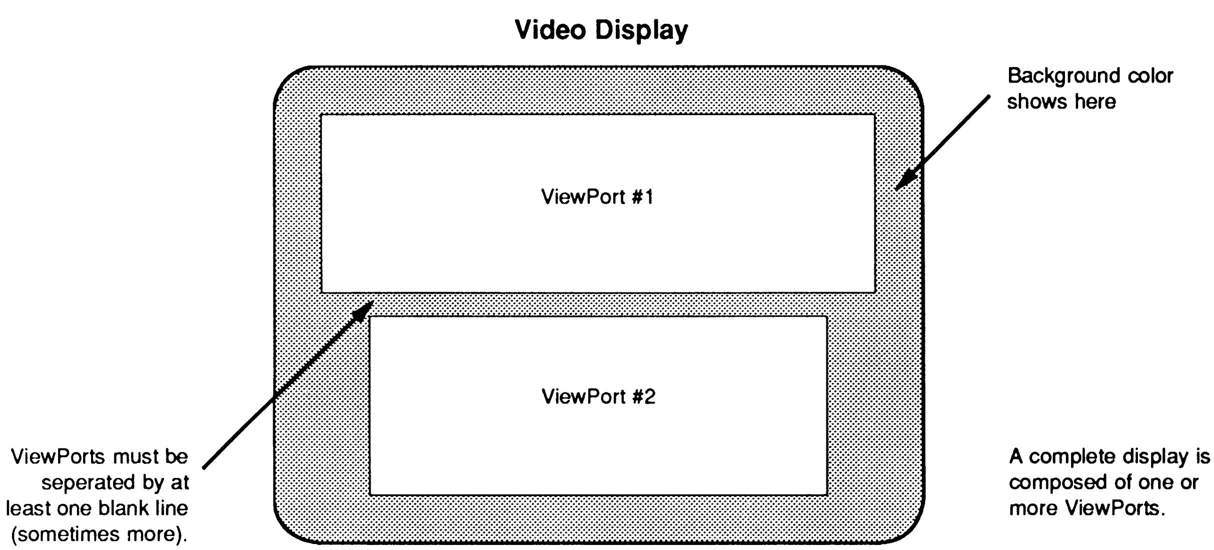


Figure 27-9: The Display Is Composed of ViewPorts

A complete display consists of one or more **ViewPorts**, whose display sections are vertically separated from each other by at least one blank scan line (non-interlaced). (If the system must make many changes to the display during the transition from one **ViewPort** to the next, there may be two or more blank scanlines between the **ViewPorts**.) The viewable area defined by each **ViewPort** is rectangular. It may be only a portion of the full **ViewPort**, it may be the full **ViewPort**, or it may be *larger* than the full **ViewPort**, allowing it to be moved within the limits of its **DisplayClip** (discussed later). You are essentially defining a display consisting of a number of stacked rectangular areas in which separate sections of graphics rasters can be shown.

LIMITATIONS ON THE USE OF VIEWPORTS

The system software for defining **ViewPorts** allows only vertically stacked fields to be defined. The following figure shows acceptable and unacceptable display configurations.

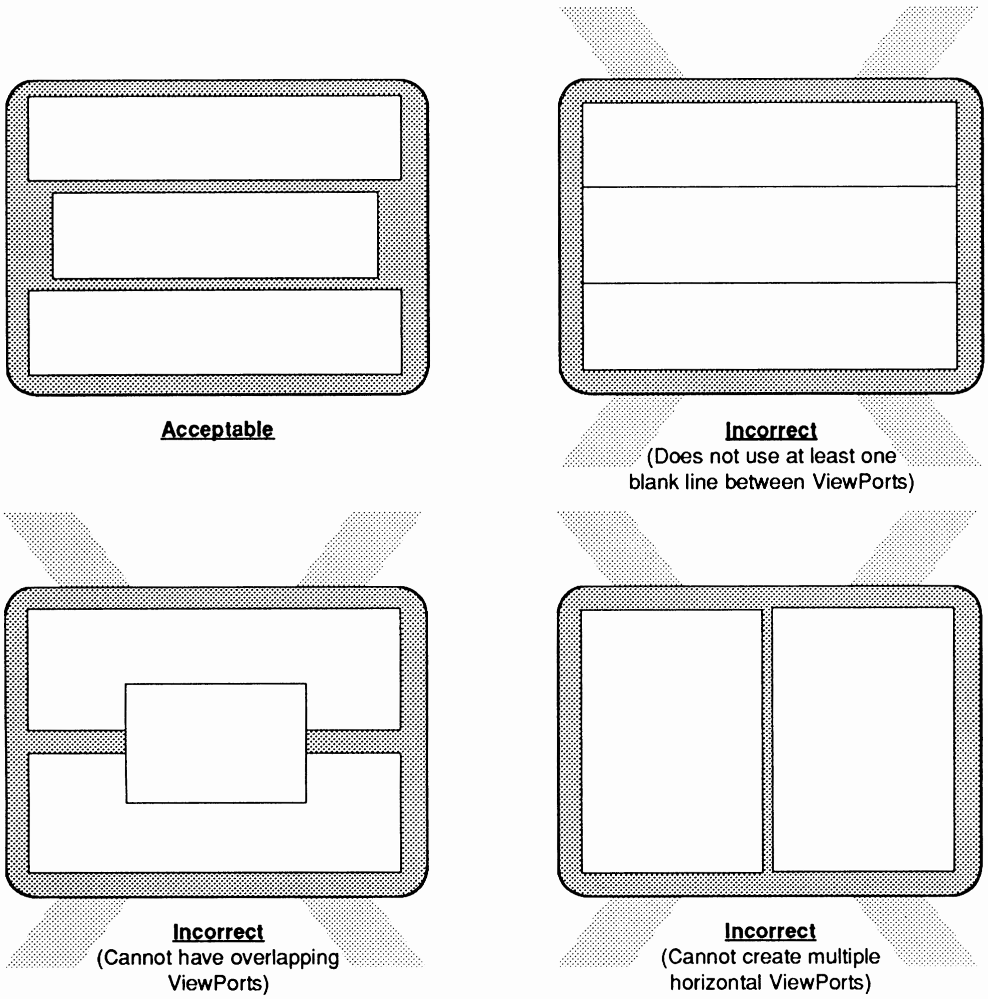


Figure 27-10: Correct and Incorrect Uses of ViewPorts

A **ViewPort** is related to the custom screen option of Intuition. In a custom screen, you can split the screen into slices as shown in the "correct" illustration of the above figure. Each custom screen can have its own set of colors, use its own resolution, and show its own display area.

CHARACTERISTICS OF A VIEWPORT

To describe a **ViewPort** fully, you need to set the following parameters: height, width, depth and display mode.

In addition to these parameters, you must tell the system the location in memory from which the data for the **ViewPort** display should be retrieved (by associating with it a **BitMap** structure) and how to position the final **ViewPort** display on the screen. The **ViewPort** will take on the user's default Workbench colors unless otherwise instructed with a **ColorMap**. See the section called "Preparing the ColorMap Structure" for more information.

VIEWPORT SIZE SPECIFICATIONS

The following figure illustrates that the variables **DHeight**, and **DWidth** specify the size of a **ViewPort**.

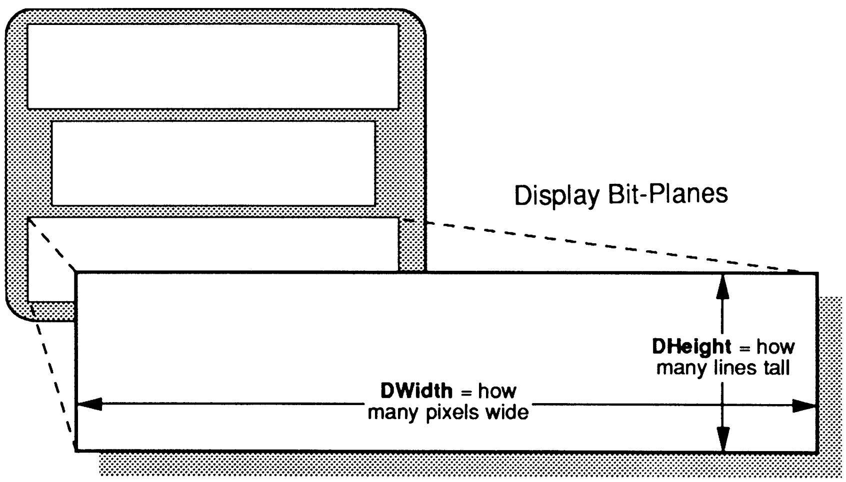


Figure 27-11: Size Definition for a ViewPort

ViewPort Height

The **DHeight** field of the **ViewPort** structure determines how many video lines will be reserved to show the height of this display segment. The size of the actual segment depends on whether you define a non-interlaced or an interlaced display. An interlaced **ViewPort** displays twice as many lines as does a non-interlaced **ViewPort** in the same physical height.

For example, a complete **View** consisting of two **ViewPorts** might be defined as follows:

- **ViewPort** 1 is 150 lines, high-resolution mode (uses the top three-quarters of the display).
- **ViewPort** 2 is 49 lines of low-resolution mode (uses the bottom quarter of the display and allows the space for the required blank line between **ViewPorts**).

Initialize the height directly in **DHeight**. Nominal height for a non-interlaced display is 200 lines for NTSC, 256 for PAL. Nominal height for an interlaced display is 400 lines for NTSC, 512 for PAL.

To set your **ViewPort** to the maximum supported (displayable) height, use the following code fragment (this requires Release 2):

```
struct DimensionInfo querydims;
struct Rectangle *oscan;
struct ViewPort viewport;

if (GetDisplayInfoData( NULL,(UBYTE *)&querydims, sizeof(struct DimensionInfo),
                        DTAG_DIMS, modeID ))
{
      /* Use StdOScan instead of MaxOScan to get standard overscan */
      /* dimensions as set by the user in Overscan Preferences     */
      oscan = &querydims.MaxOScan;
      viewPort->DHeight = oscan->MaxY - oscan->MinY + 1;
}
```

ViewPort Width

The **DWidth** variable in the **ViewPort** structure determines how wide, in pixels, the display segment will be. To set your **ViewPort** to the maximum supported (displayable) NTSC high-resolution width, use the following fragment (this requires Release 2):

```
struct DimensionInfo querydims;
struct Rectangle *oscan;
struct ViewPort viewport;

/* Use PAL_MONITOR_ID instead of NTSC_MONITOR_ID to get PAL dimensions */
if (GetDisplayInfoData( NULL,(UBYTE *)&querydims, sizeof(querydims),
      DTAG_DIMS, NTSC_MONITOR_ID|HIRES_KEY ))
{
      /* Use StdOScan instead of MaxOScan to get standard overscan */
      /* dimensions as set by the user in Overscan Preferences     */
      oscan = &querydims.MaxOScan;
      viewPort->DWidth = oscan->MaxX - oscan->MinX + 1;
}
```

You may specify a smaller value of pixels per line to produce a narrower display segment or simply set **ViewPort.DWidth** to the nominal value for this resolution.

Although the system software allows you define low-resolution displays as wide as 362 pixels and high-resolution displays as wide as 724 pixels, you should use caution in exceeding the normal values of 320 or 640, respectively. Because display overscan varies from one monitor to another, many video displays will not be able to show all of a wider display, and sprite display may also be affected. However, if you use the standard overscan values (**DimensionInfo.StdOScan**) provided by the Release 2 function **GetDisplayInfoData()** as shown above, the user's preference for the size of the display will be satisfied.

If you are using hardware sprites or VSprites with your display, and you specify **ViewPort** widths exceeding 320 or 640 pixels (for low or high-resolution, respectively), it is likely that some hardware sprites will not be properly rendered on the screen. These sprites may not be rendered because playfield DMA (direct memory access) takes precedence over sprite DMA when an extra-wide display is produced. See the *Amiga Hardware Reference Manual* for a more complete description of this phenomenon.

VIEWPORT COLOR SELECTION

The maximum number of colors that a **ViewPort** can display is determined by the depth of the **BitMap** that the **ViewPort** displays. The depth is specified when the **BitMap** is initialized. See the section below called ''Preparing the BitMap Structure.''

Depth determines the number of bitplanes used to define the colors of the rectangular image you are trying to build (the raster image) and the number of different colors that can be displayed at the same time within a **ViewPort**. For any single pixel, the system can display any one of 4,096 possible colors.

The following table shows depth values and the corresponding number of possible colors for each value.

Table 27-1: Depth Values and Number of Colors in the ViewPort

Colors	Depth Value	
2	1	
4	2	
8	3	(Note 1)
16	4	(Notes 1,2)
32	5	(Notes 1,2,3)
16	6	(Notes 1,4)
64	6	(Notes 1,2,3,5)
4,096	6	(Notes 1,2,3,6)

Notes:
1. Not available for SUPERHIRES.
2. Single-playfield mode only - DUALPF *not* one of the **ViewPort**'s attributes.
3. Low-resolution mode only - neither HIRES nor SUPERHIRES one of the **ViewPort** attributes.
4. Dual Playfield mode - DUALPF is an attribute of this **ViewPort**. Up to eight colors (in three planes) for each playfield.
5. Extra-Half-Brite mode - EXTRA_HALFBRITE is an attribute of this **ViewPort**.
6. Hold-And-Modify mode only - HAM is an attribute of this **ViewPort**.

The color palette used by a **ViewPort** is specified in a **ColorMap**. See the section called "Preparing the ColorMap" for more information.

Depending on whether single- or dual-playfield mode is used, the system will use different color register groupings for interpreting the on-screen colors. The table below details how the depth and the different **ViewPort** modes affect the registers the system uses.

Table 27-2: Color Registers Used in Single-playfield Mode

Depth	Color Registers Used	
1	0,1	
2	0-3	
3	0-7	
4	0-15	
5	0-31	
6	0-31	(if EXTRA_HALFBRITE is an attribute of this **ViewPort**.)
6	0-15	(if HAM is an attribute of this **ViewPort**.)

The following table shows the five possible combinations when DUALPF *is* an attribute of the **ViewPort**.

Table 27-3: Color Register Used in Dual-playfield Mode

Depth (PF-1)	Color Registers	Depth (PF-2)	Color Registers
1	0,1	1	8,9
2	0-3	1	8,9
2	0-3	2	8-11
3	0-7	2	8-11
3	0-7	3	8-15

VIEWPORT DISPLAY MODES

The system has many different display modes that you can specify for each **ViewPort**. Under 1.3, the eight constants that control the modes are DUALPF, PFBA, HIRES, SUPERHIRES, LACE, HAM, SPRITES, and EXTRA_HALFBRITE. Some, but not all of the modes can be combined in a **ViewPort**. HIRES and LACE combine to make a high-resolution, interlaced **ViewPort**, but HIRES and SUPERHIRES conflict, and cannot be combined.

Under 1.3, you set these flags directly in the **Modes** field during initialization of the **ViewPort**. Under Release 2, there are many more display modes possible than in 1.3 so a new system of flags and structures is used to set the mode. With Release 2, you set the display mode for a **ViewPort** by using the **VideoControl()** function as described in the section on ''Monitors, Modes and the Display Database'' later in this chapter.

The DUALPF and PFBA modes are related. DUALPF tells the system to treat the raster specified by this **ViewPort** as the first of two independent and separately controllable playfields. It also modifies the manner in which the pixel colors are selected for this raster (see the above table).

When PFBA is specified, it indicates that the second playfield has video priority over the first one. Playfield relative priorities can be controlled when the playfield is split into two overlapping regions. Single-playfield and dual-playfield modes are discussed in ''Advanced Topics'' below.

HIRES tells the system that the raster specified by this **ViewPort** is to be displayed with (nominally) 640 horizontal pixels, rather than the 320 horizontal pixels of Lores mode.

SUPERHIRES tells the system that the raster specified by this **ViewPort** is to be displayed with (nominally) 1280 horizontal pixels. This can be used with 31 kHz scan rates to provide the VGA and Productivity modes available in Release 2. SUPERHIRES modes require both the ECS and Release 2. See the section on ''Determining Chip Versions'' earlier in this chapter for an explanation of how to find out if the ECS is present.

LACE tells the system that the raster specified by this **ViewPort** is to be displayed in interlaced mode. If the **ViewPort** is non-interlaced and the **View** is interlaced, the **ViewPort** will be displayed at its specified height and will look only slightly different than it would look when displayed in a non-interlaced **View** (this is handled by the system automatically). See ''Interlaced Mode vs. Non-interlaced Mode'' below for more information.

HAM tells the system to use "hold-and-modify" mode, a special mode that lets you display up to 4,096 colors on screen at the same time. It is described in the "Advanced Topics" section.

SPRITES tells the system that you are using sprites in this display (either VSprites or Simple Sprites). The system will load color registers for the sprites. Note that since the mouse pointer is a sprite, omitting this mode will prevent the mouse pointer from being displayed when this **ViewPort** is frontmost. See the "Graphics Sprites, Bobs and Animation" chapter for more information about sprites.

EXTRA_HALFBRITE tells the system to use the Extra-Half-Brite mode, a special mode that allows you to display up to 64 colors on screen at the same time. It is described in the "Advanced Topics" section.

If you peruse the *<graphics/view.h>* include file you will see another flag, EXTENDED_MODE. Never set this flag yourself; it is used by the Release 2 system to control more advanced mode features.

Be sure to read the section on "Monitors, Modes and the Display Database" for additional information about the **ViewPort** mode and how it has changed in the Release 2 version of the operating system.

Single-playfield Mode vs. Dual-playfield Mode

When you specify single-playfield mode you are asking that the system treat all bitplanes as part of the definition of a single playfield image. Each of the bitplanes defined as part of this **ViewPort** contributes data bits that determine the color of the pixels in a single playfield.

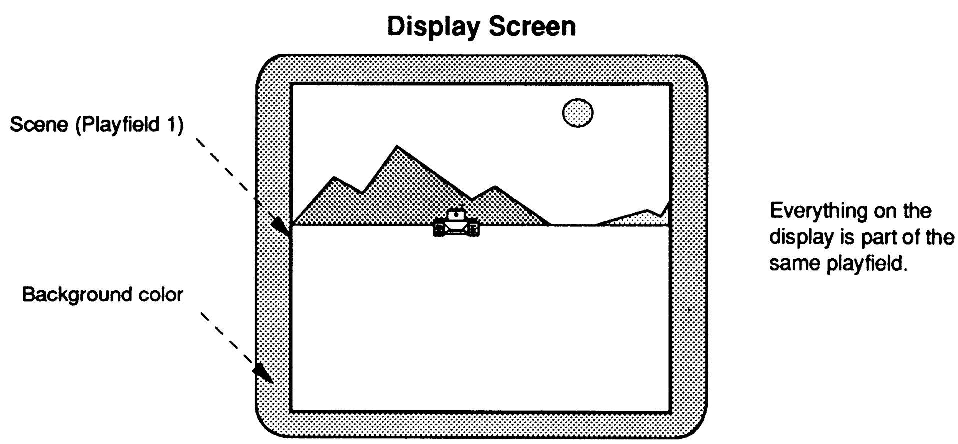


Figure 27-12: A Single-playfield Display

If you use dual-playfield mode, you can define two independent, separately controllable playfield areas as shown on the next page.

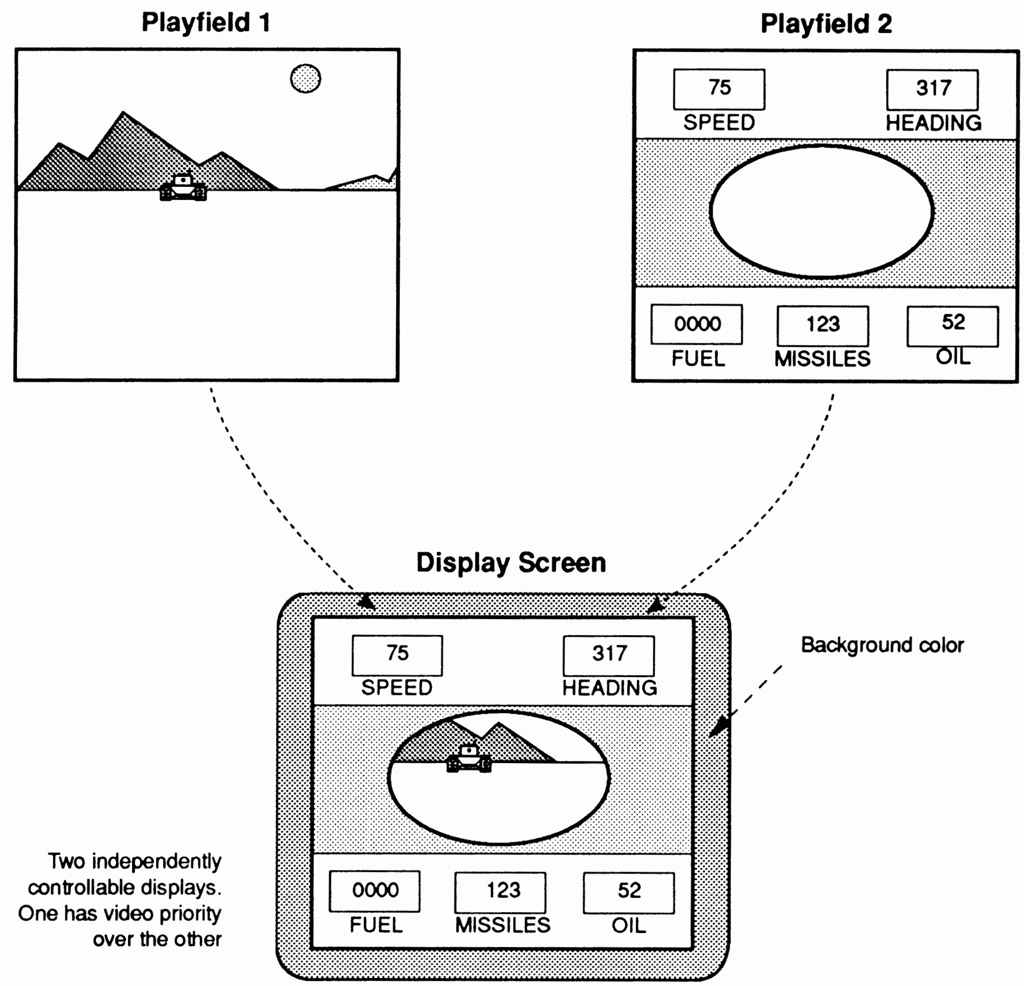


Figure 27-13: A Dual-playfield Display

In the previous figure, PFBA was included in the display mode. If PFBA had not been included, the relative priorities would have been reversed; playfield 2 would have appeared to be behind playfield 1.

Low-resolution Mode vs. High-resolution Mode

In LORES mode, horizontal lines of 320 pixels fill most of the ordinary viewing area. The system software lets you define a screen segment width up to 362 pixels in this mode, or you can define a screen segment as narrow as you desire (minimum of 16 pixels). In HIRES mode, 640 pixels fill a horizontal line. In this mode you can specify any width from 16 to 724 pixels. In SUPERHIRES mode, 1280 pixels fill a horizontal line. In this mode you can specify any width from 16 to 1448 pixels. The fact that many monitor manufacturers set their monitors to overscan the video display normally limits you to showing only 16 to 320 pixels per line in LORES, 16 to 640 pixels per line in HIRES, or 16 to 1280 pixels per line in SUPERHIRES. Under Release 2, the user can set the monitor's viewable screen size with the Preferences Overscan editor.

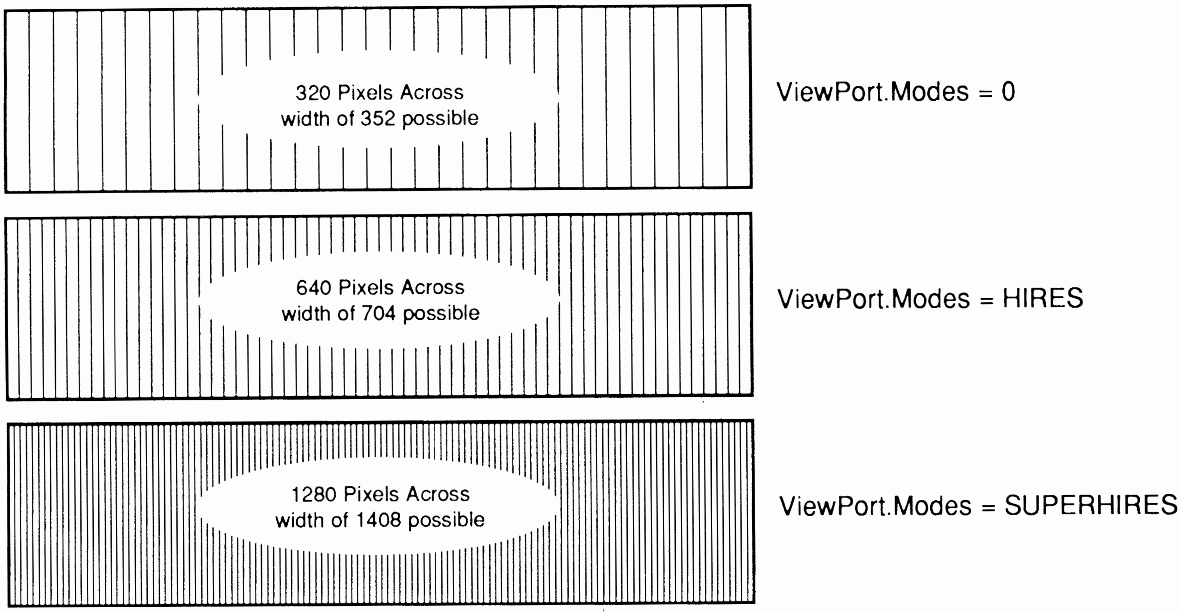


Figure 27-14: How HIRES and SUPERHIRES Affect the Width of Pixels

Interlaced Mode vs. Non-interlaced Mode

In interlaced mode, there are twice as many lines available as in non-interlaced mode, providing better vertical resolution in the same display area.

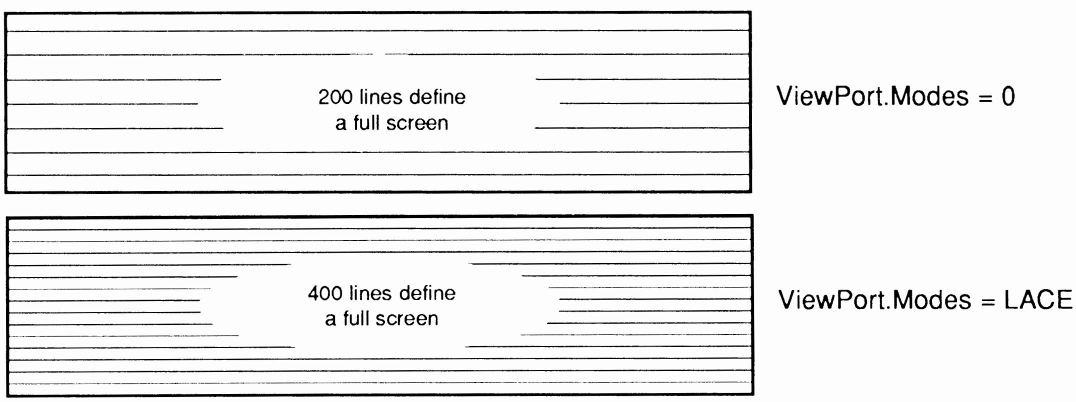


Figure 27-15: How LACE Affects Vertical Resolution

If the **View** structure does not specify LACE, and the **ViewPort** specifies LACE, only the top half of the **ViewPort** data will be displayed. If the **View** structure specifies LACE and the **ViewPort** is non-interlaced, the same **ViewPort** data will be repeated in both fields. The height of the **ViewPort** display is the height specified in the **ViewPort** structure. If both the **View** and the **ViewPort** are interlaced, the **ViewPort** will be built with double the normal vertical resolution. That means it will need twice as much data space in memory as a non-interlaced picture to fill the display.

VIEWPORT DISPLAY MEMORY

The picture you create in memory can be larger than the screen image that can be displayed within your **ViewPort**. This big picture (called a raster and represented by the **BitMap** structure) can have a maximum size dependent upon the version of the Agnus chip in the Amiga. The ECS Agnus can handle rasters up to 16,384 by 16,384 pixels. Older Agnus chips are limited to rasters up to 1,024 by 1,024 pixels. The section on ''Determining Chip Versions'' earlier in this chapter explains how to find out which Agnus is installed.

The example in the following figure introduces terms that tell the system how to find the display data and how to display it in the **ViewPort**. These terms are RHeight, RWidth, **RyOffset**, **RxOffset**, **DHeight**, **DWidth**, **DyOffset** and **DxOffset**.

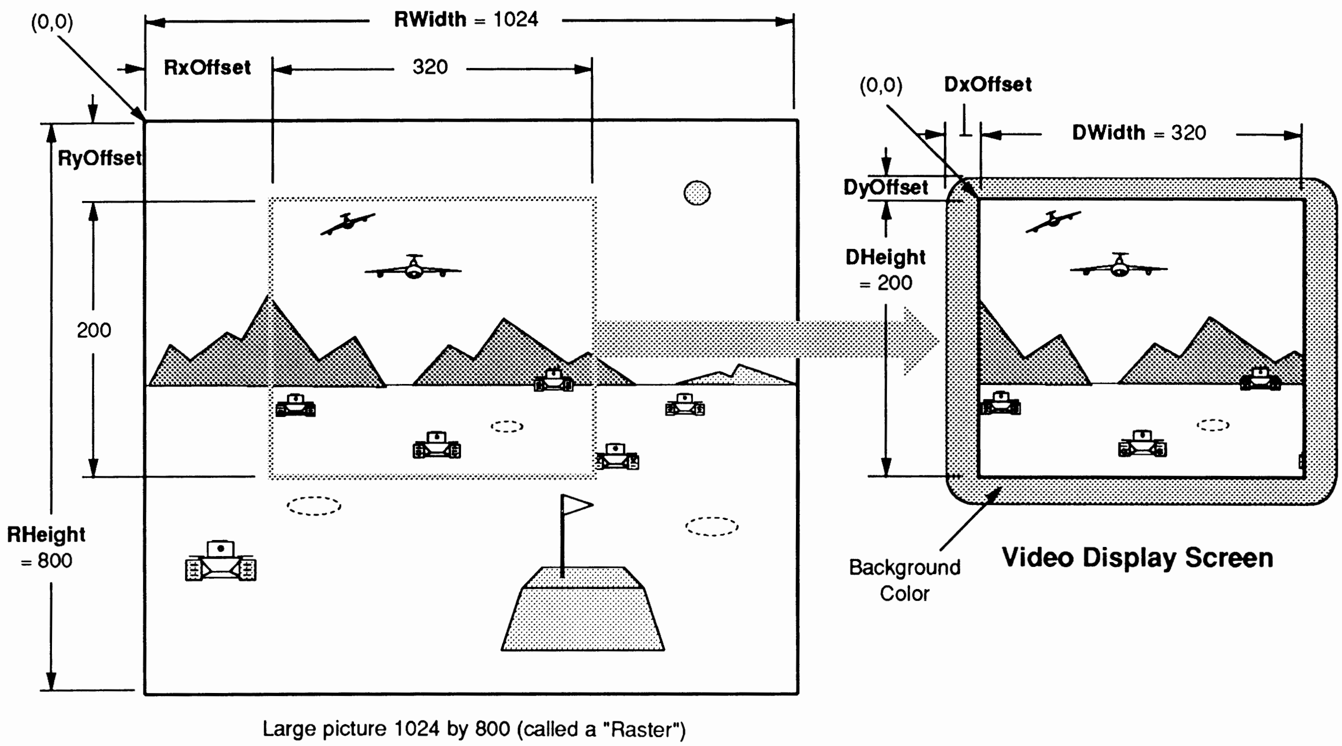


Figure 27-16: ViewPort Data Area Parameters

The terms RHeight and RWidth do not appear in actual system data structures. They refer to the dimensions of the raster and are used here to relate the size of the raster to the size of the display area. RHeight is the number of rows in the raster and RWidth is the number of bytes per row times 8. The raster shown in the figure is too big to fit entirely in the display area, so you tell the system which pixel of the raster should appear in the upper left corner of the display segment specified by your **ViewPort**. The variables that control that placement are **RyOffset** and **RxOffset**.

To compute **RyOffset** and **RxOffset**, you need RHeight, RWidth, **DHeight**, and **DWidth**. The **DHeight** and **DWidth** variables define the height and width in pixels of the portion of the display that you want to appear in the **ViewPort**. The example shows a full-screen, low-resolution mode (320-pixel), non-interlaced (200-line) display formed from the larger overall picture.

Normal values for **RyOffset** and **RxOffset** are defined by the formulas:

0 < = **RyOffset** < = (RHeight - **DHeight**)
0 < = **RxOffset** < = (RWidth - **DWidth**)

Once you have defined the size of the raster and the section of that raster that you wish to display, you need only specify where to put this **ViewPort** on the screen. This is controlled by the **ViewPort** variables **DyOffset** and **DxOffset**. These are offsets relative to the **View.DxOffset** and **DyOffset**. Possible NTSC values for **DyOffset** range from -23 to +217 (-46 to +434 if the **ViewPort** is interlaced), PAL values range from -15 to +267 (-30 to +534 for interlaced **ViewPorts**). Possible values for **DxOffset** range from -18 to +362 (-36 to +724 if the **ViewPort** is Hires, -72 to +1448 if SuperHires), when the **View** is in its default, initialized position.

The parameters shown in the figure above are distributed in the following data structures:

- **View** (information about the whole display) includes the variables that you use to position the whole display on the screen. The **View** structure contains a **Modes** field used to determine if the whole display is to be interlaced or non-interlaced. It also contains pointers to its list of **ViewPorts** and pointers to the Copper instructions produced by the system to create the display you have defined.

- **ViewPort** (information about this segment of the display) includes the values **DxOffset** and **DyOffset** that are used to position this portion relative to the overall **View**. The **ViewPort** also contains the variables **DHeight** and **DWidth**, which define the size of this display segment; a **Modes** variable; and a pointer to the local **ColorMap**. Under Release 2, the **VideoControl()** function and its various tags are used to manipulate the **ColorMap** and **ViewPort.Modes**. Each **ViewPort** also contains a pointer to the next **ViewPort**. You create a linked list of **ViewPorts** to define the complete display.

- **RasInfo** (information about the raster) contains the variables **RxOffset** and **RyOffset**. It also contains pointers to the **BitMap** structure and to a companion **RasInfo** structure if this is a dual playfield.

- **BitMap** (information about memory usage) tells the system where to find the display and drawing area memory and shows how this memory space is organized, including the display's depth.

You must allocate enough memory for the display you define. The memory you use for the display may be shared with the area control structures used for drawing. This allows you to draw into the same areas that you are currently displaying on the screen.

As an alternative, you can define two **BitMaps**. One of them can be the active structure (that being displayed) and the other can be the inactive structure. If you draw into one **BitMap** while displaying another, the user cannot see the drawing taking place. This is called *double-buffering* of the display. See "Advanced Topics" below for an explanation of the steps required for double-buffering. Double-buffering takes twice as much memory as single-buffering because two full displays are produced.

To determine the amount of required memory for each **ViewPort** for single-buffering, you can use the following formula.

```
#include <graphics/gfx.h>

/* Depth, Width, and Height get set to something reasonable. */
UBYTE Depth, Width, Height;

/* Calculate resulting VP size. */
bytes_per_ViewPort = Depth * RASSIZE(Width, Height);
```

RASSIZE() is a system macro attuned to the current design of the system memory allocation for display rasters. See the *<graphics/gfx.h>* include file for the formula with which **RASSIZE()** is calculated.

For example, a 32-color **ViewPort** (depth = 5), 320 pixels wide by 200 lines high currently uses 40,000 bytes. A 16-color **ViewPort** (depth = 4), 640 pixels wide by 400 lines high currently uses 128,000 bytes.

FORMING A BASIC DISPLAY

Here are the data structures that you need to define to create a basic display:

```
struct View view;             /* These get used in all versions of the OS */
struct ViewPort viewPort;
struct BitMap bitMap;
struct RasInfo rasInfo;
struct ColorMap *cm;

struct ViewExtra *vextra;       /* Extra View data, new in Release 2     */
struct ViewPortExtra *vpextra;  /* Extra ViewPort data, new in Release 2 */
struct MonitorSpec *monspec;    /* Monitor data needed in Release 2      */
struct DimensionInfo dimquery;  /* Display dimension data needed in Release 2 */
```

ViewExtra and **ViewPortExtra** are new data structures used in Release 2 to hold extended data about their corresponding parent structure. **ViewExtra** contains information about the video monitor being used to render the **View**. **ViewPortExtra** contains information required for clipping of the **ViewPort**.

GfxNew() is used to create these extended data structures and **GfxAssociate()** is used to associate the extended data structure with an appropriate parent structure. Although **GfxAssociate()** can associate a **ViewPortExtra** structure with a **ViewPort**, it is better to use **VideoControl()** with the VTAG_VIEWPORTEXTRA_SET tag instead. Keep in mind that **GfxNew()** allocates memory for the resulting data structure which must be returned using **GfxFree()** before the application exits. The function **GfxLookUp()** will find the address of an extended data structure from the address of its parent.

Preparing the View Structure

The following code prepares the **View** structure for further use:

```
InitView(&view);       /*  Initialize the View.  */
view.Modes |= LACE;    /*  Only interlaced, 1.3 displays require this */
```

For Release 2 applications, a **ViewExtra** structure must also be created with **GfxNew()** and associated with this **View** with **GfxAssociate()** as shown in the example programs RGBBoxes.c and WBClone.c.

```
/* Form the ModeID from values in <displayinfo.h> */
modeID=DEFAULT_MONITOR_ID | HIRESLACE_KEY;

/* Make the ViewExtra structure */
if( vextra=GfxNew(VIEW_EXTRA_TYPE) )
    {
    /* Attach the ViewExtra to the View */
    GfxAssociate(&view , vextra);
    view.Modes |= EXTEND_VSTRUCT;

    /* Initialize the MonitorSpec field of the ViewExtra */
    if( monspec=OpenMonitor(NULL,modeID) )
        vextra->Monitor=monspec;
    else
        fail("Could not get MonitorSpec\n");
    }
 else fail("Could not get ViewExtra\n");
```

Preparing the BitMap Structure

The **BitMap** structure tells the system where to find the display and drawing memory and how this memory space is organized. The following code section prepares a **BitMap** structure, including allocation of memory for the bitmap. This is done with two functions, **InitBitMap()** and **AllocRaster()**. **InitBitMap()** takes four arguments—a pointer to a **BitMap** and the depth, width, and height of the desired bitmap. Once the bitmap is initialized, memory for its bitplanes must be allocated. **AllocRaster()** takes two arguments—width and height. Here is a code section to initialize a bitmap:

```
/*  Init BitMap for RasInfo.  */
InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);

/* Set the plane pointers to NULL so the cleanup routine will know if they were used. */
for(depth=0; depth<DEPTH; depth++)
    bitMap.Planes[depth] = NULL;

/*  Allocate space for BitMap.  */
for(depth=0; depth<DEPTH; depth++)
    {
    bitMap.Planes[depth] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
    if (bitMap.Planes[depth] == NULL)
        cleanExit(RETURN_WARN);
    }
```

This code allocates enough memory to handle the display area for as many bitplanes as the depth you have defined.

Preparing the RasInfo Structure

The **RasInfo** structure provides information to the system about the location of the **BitMap** as well as the positioning of the display area as a window against a larger drawing area. Use the following steps to prepare the **RasInfo** structure:

```
/* Initialize the RasInfos. */
rasInfo.BitMap = &bitMap;   /* Attach the corresponding BitMap.         */
rasInfo.RxOffset = 0;       /* Align upper left corners of display      */
rasInfo.RyOffset = 0;       /*   with upper left corner of drawing area. */
rasInfo.Next = NULL;        /* for a single playfield display, there
                             * is only one RasInfo structure present    */
```

The system may be made to reinterpret the **RxOffset** and **RyOffset** values in a **ViewPort**'s **RasInfo** structure by calling **ScrollVPort()** with the address of the **ViewPort**. Changing one or both offsets and calling **ScrollVPort()** has the effect of scrolling the **ViewPort**.

Preparing the ViewPort Structure

To prepare the **ViewPort** structure for further use, you call **InitVPort()** and initialize certain fields as follows:

```
InitVPort(&viewPort);          /* Initialize the ViewPort.           */
viewPort.RasInfo = &rasInfo;   /* The rasInfo must also be initialized */
viewPort.DWidth  = WIDTH;
viewPort.DHeight = HEIGHT;

/* Under 1.3, you should set viewPort.Modes here to select a display mode. */
/* Under Release 2, use VideoControl() with VTAG_NORMAL_DISP_SET to select */
/* a display mode by attaching a DisplayInfo structure to the ViewPort.    */
```

The **InitVPort()** routine presets certain default values in the **ViewPort** structure. The defaults include:

- **Modes** variable set to zero—this means you select a low-resolution display. (To alter this, use **VideoControl()** with the VTAG_NORMAL_DISP_SET tag as explained below.)
- **Next** variable set to NULL—no other **ViewPort** is linked to this one. If you want a display with multiple **ViewPorts**, you must fill in the link yourself.

If you want to create a **View** with two or more **ViewPorts** you must declare and initialize the **ViewPorts** as above. Then link them together using the **ViewPort.Next** field with a NULL link for the **ViewPort** at the end of the chain:

```
viewPortA.Next = &viewPortB;    /* Tell first one the address of the second. */
viewPortB.Next = NULL;          /* There are no others after this one. */
```

For Release 2 applications, once a **ViewPort** has been prepared, a **ViewPortExtra** structure must also be created with **GfxNew()**, initialized, and associated with the **ViewPort** via the **VideoControl()** function. In addition, a **DisplayInfo** for this mode must be attached to the **ViewPort**. The fragment below shows how to do this. For complete examples, refer to the program listings of RGBBoxes.c and WBClone.c.

```
struct TagItem vcTags[] =               /* These tags will be passed to the  */
{                                       /* VideoControl() function to set up */
    { VTAG_ATTACH_CM_SET, NULL },       /* the extended ViewPort structures  */
    { VTAG_VIEWPORTEXTRA_SET, NULL },   /* required in Release 2.  The NULL  */
    { VTAG_NORMAL_DISP_SET, NULL },     /* ti_Data field of these tags must  */
    { VTAG_END_CM, NULL }               /* be filled in before making the    */
};                                      /* call to VideoControl().           */

struct DimensionInfo dimquery; /* Release 2 structure for display size data */

/* Make a ViewPortExtra and get ready to attach it */
if( vpextra = GfxNew(VIEWPORT_EXTRA_TYPE) )
    {
    vcTags[1].ti_Data = (ULONG) vpextra;

    /* Initialize the DisplayClip field of the ViewPortExtra structure */
    if( GetDisplayInfoData( NULL , (UBYTE *) &dimquery ,
                           sizeof(struct dimquery) , DTAG_DIMS, modeID) )
        {
        vpextra->DisplayClip = dimquery.Nominal;

        /* Make a DisplayInfo and get ready to attach it */
        if( !(vcTags[2].ti_Data = (ULONG) FindDisplayInfo(modeID)) )
            fail("Could not get DisplayInfo\n");
        }
    else fail("Could not get DimensionInfo\n");
    }
else fail("Could not get ViewPortExtra\n");

/* This is for backwards compatibility with, for example,   */
/* a 1.3 screen saver utility that looks at the Modes field */
viewPort.Modes = (UWORD) (modeID & 0x0000ffff);
```

Preparing the ColorMap Structure

When the **View** is created, Copper instructions are generated to change the current contents of each color register just before the topmost line of a **ViewPort** so that this **ViewPort**'s color registers will be used for interpreting its display. To set the color registers you create a **ColorMap** for the **ViewPort** with **GetColorMap()** and call **SetRGB4()**. Here are the steps used in 1.3 to initialize a **ColorMap**:

```
if( view.ColorMap=GetColorMap( 4L ) )
    LoadRGB4((&viewPort, colortable, 4);
```

Under Release 2, a **ColorMap** is attached to the **View**— usually along with **DisplayInfo** and **ViewExtra**—by calling the **VideoControl()** function.

```
/*  RGB values for the four colors used.  */
#define BLACK 0x000
#define RED   0xf00
#define GREEN 0x0f0
#define BLUE  0x00f

/*  Define some colors in an array of UWORDS.  */
static UWORD colortable[] = { BLACK, RED, GREEN, BLUE };

/* Fill the TagItem Data field with the address of the properly initialized
   (including ViewPortExtra) structure to be passed to VideoControl().     */
vc[0].ti_Data = (ULONG)viewPort;

/* Init ColorMap.  2 planes deep, so 4 entries (2 raised to #planes power). */
if(cm = GetColorMap( 4L ) )
{
    /* For applications that must be compatible with 1.3, replace the next 2 */
    /* lines with: viewPort.ColorMap=cm;                                     */
    if( VideoControl( cm , vcTags ) )
        fail("Could not attach extended structures\n");

    /*  Change colors to those in colortable.  */
    LoadRGB4(&viewPort, colortable, 4);
}
```

> *The 4 Is For Bits, Not Entries.* The **4** in the name **LoadRGB4()** refers to the fact that each of the red, green, and blue values in a color table entry consists of four bits. It has nothing to do with the fact that this particular color table contains four entries. The call **GetRGB4()** returns the RGB value of a single entry of a **ColorMap**. **SetRGB4CM()** allows individual control of the entries in the **ColorMap** before or after linking it into the **ViewPort**.

The **LoadRGB4()** call above could be replaced with the following:

```
register USHORT entry;

/*  Operate on the same four ColorMap entries as above.  */
for (entry = 0; entry < 4; entry++)
    {
    /*  Call SetRGB4CM() with the address of the ColorMap, the entry to be
        changed, and the Red, Green, and Blue values to be stored there.
    */
    SetRGB4CM(viewPort.ColorMap, entry,
    /*  Extract the three color values from the one colortable entry.  */
        ((colortable[entry] & 0x0f00) >> 8),
            ((colortable[entry] & 0x00f0) >> 4),
                (colortable[entry] & 0x000f));
    }
```

Notice above how the four bits for each color are masked out and shifted right to get values from 0 to 15.

> *WARNING!* It is important to use *only* the standard system **ColorMap**-related calls to access the **ColorMap** entries. These calls will remain compatible with recent and future enhancements to the **ColorMap** structure.

You might need to specify more colors in the color map than you think. If you use a dual playfield display (covered later in this chapter) with a depth of 1 for each of the two playfields, this means a total of four colors (two for each playfield). However, because playfield 2 uses color registers starting from number 8 on up when in dual-playfield mode, the color map must be initialized to contain at least 10 entries. That is, it must contain entries for colors 0 and 1 (for playfield 1) and color numbers 8 and 9 (for playfield 2). Space for sprite colors must be allocated as well. For Amiga system software version 1.3 and earlier, ***when in doubt***, allocate a **ColorMap** with 32 entries, just in case.

Creating the Display Instructions

Now that you have initialized the system data structures, you can request that the system prepare a set of display instructions for the Copper using these structures as input data. During the one or more blank vertical lines that precede each **ViewPort**, the Copper is busy changing the characteristics of the display hardware to match the characteristics you expect for this **ViewPort**. This may include a change in display resolution, a change in the colors to be used, or other user-defined modifications to system registers.

Here is the code that creates the display instructions:

```
/*  Construct preliminary Copper instruction list.  */
MakeVPort( &view, &viewPort );
```

In this line of code, **&view** is the address of the **View** structure and **&viewPort** is the address of the first **ViewPort** structure. Using these structures, the system has enough information to build the instruction stream that defines your display.

MakeVPort() creates a special set of instructions that controls the appearance of the display. If you are using animation, the graphics animation routines create a special set of instructions to control the hardware sprites and the system color registers. In addition, the advanced user can create special instructions (called user Copper instructions) to change system operations based on the position of the video beam on the screen.

All of these special instructions must be merged together before the system can use them to produce the display you have designed. This is done by the system routine **MrgCop()** (which stands for "Merge Coprocessor Instructions"). Here is a typical call:

```
/*  Merge preliminary lists into a real Copper list in the view structure.  */
MrgCop( &view );
```

LOADING AND DISPLAYING THE VIEW

To display the **View**, you need to load it using **LoadView()** and turn on the direct memory access (DMA). A typical call is shown below.

```
LoadView(&view);
```

The **&view** argument is the address of the **View** structure defined in the example above.

There are two macros, defined in *<graphics/gfxmacros.h>*, that control display DMA: **ON_DISPLAY** and **OFF_DISPLAY**. They simply turn the display DMA control bit in the DMA control register on or off.

If you are drawing to the display area and do not want the user to see intermediate steps in the drawing, you can turn off the display. Because **OFF_DISPLAY** shuts down the display DMA and possibly speeds up other system operations, it can be used to provide additional memory cycles to the blitter or the 68000. The distribution of system DMA, however, allows four-channel sound, disk read/write, and a sixteen-color, low-resolution display (or four-color, high-resolution display) to operate at the same time with no slowdown (7.1 megahertz effective rate) in the operation of the 68000. Using **OFF_DISPLAY** in a multitasking environment may, however, be an unfriendly thing to do to the other running processes. Use **OFF_DISPLAY** with discretion.

A Custom ViewPort Example

The following example creates a **View** consisting of one **ViewPort** set to an NTSC, high-resolution, interlaced display mode of nominal dimensions. This example shows both the old 1.3 way of setting up the **ViewPort** and the new method used in Release 2.

```
;/*  RGBBoxes.c simple ViewPort example -- works with 1.3 and Release 2
LC -b1 -cfistq -v -y -j73 RGBBoxes.c
Blink FROM LIB:c.o,RGBBoxes.o TO RGBBoxes LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <graphics/view.h>
#include <graphics/displayinfo.h>
#include <graphics/gfxnodes.h>
#include <graphics/videocontrol.h>
#include <libraries/dos.h>
#include <utility/tagitem.h>

#include <clib/graphics_protos.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

#include <stdio.h>
#include <stdlib.h>

#define DEPTH 2      /*  The number of bitplanes.  */
#define WIDTH 640    /*  Nominal width and height  */
#define HEIGHT 400   /*  used in 1.3.              */

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

VOID drawFilledBox(WORD , WORD );  /* Function prototypes */
VOID cleanup(int );
VOID fail(STRPTR);

struct GfxBase *GfxBase = NULL;

/*  Construct a simple display.  These are global to make freeing easier.   */
struct View view, *oldview=NULL;  /*  Pointer to old View we can restore it.*/
struct ViewPort viewPort = { 0 };
struct BitMap bitMap = { 0 };
struct ColorMap *cm=NULL;

struct ViewExtra *vextra=NULL;      /* Extended structures used in Release 2 */
struct MonitorSpec *monspec=NULL;
struct ViewPortExtra *vpextra=NULL;
struct DimensionInfo dimquery = { 0 };

UBYTE *displaymem = NULL;      /*  Pointer for writing to BitMap memory.  */

#define BLACK 0x000            /*  RGB values for the four colors used.   */
#define RED   0xf00
#define GREEN 0x0f0
#define BLUE  0x00f

/*
 * main():  create a custom display; works under either 1.3 or Release 2
 */
VOID main(VOID)
{
WORD depth, box;
struct RasInfo rasInfo;
ULONG modeID;
```

```
struct TagItem vcTags[] =
{
    {VTAG_ATTACH_CM_SET, NULL },
    {VTAG_VIEWPORTEXTRA_SET, NULL },
    {VTAG_NORMAL_DISP_SET, NULL },
    {VTAG_END_CM, NULL }
};

/*  Offsets in BitMap where boxes will be drawn.  */
static SHORT boxoffsets[] = { 802, 2010, 3218 };

static UWORD colortable[] = { BLACK, RED, GREEN, BLUE };

/* Open the graphics library */
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33L);
if(GfxBase == NULL)
    fail("Could not open graphics library\n");

/*  Example steals screen from Intuition if Intuition is around.      */
oldview = GfxBase->ActiView;   /* Save current View to restore later. */

InitView(&view);           /*  Initialize the View and set View.Modes.     */
view.Modes |= LACE;        /*  This is the old 1.3 way (only LACE counts). */

if(GfxBase->LibNode.lib_Version >= 36)
    {
    /* Form the ModeID from values in <displayinfo.h> */
    modeID=DEFAULT_MONITOR_ID | HIRESLACE_KEY;

    /*  Make the ViewExtra structure   */
    if( vextra=GfxNew(VIEW_EXTRA_TYPE) )
        {
        /* Attach the ViewExtra to the View */
        GfxAssociate(&view , vextra);
        view.Modes |= EXTEND_VSTRUCT;

        /* Create and attach a MonitorSpec to the ViewExtra */
        if( monspec=OpenMonitor(NULL,modeID) )
            vextra->Monitor=monspec;
        else
            fail("Could not get MonitorSpec\n");
        }
     else fail("Could not get ViewExtra\n");
     }


/*  Initialize the BitMap for RasInfo.  */
InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);

/* Set the plane pointers to NULL so the cleanup routine */
/* will know if they were used.                          */
for(depth=0; depth<DEPTH; depth++)
    bitMap.Planes[depth] = NULL;

/*  Allocate space for BitMap.             */
for (depth=0; depth<DEPTH; depth++)
    {
    bitMap.Planes[depth] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
    if (bitMap.Planes[depth] == NULL)
        fail("Could not get BitPlanes\n");
    }

rasInfo.BitMap = &bitMap;       /*  Initialize the RasInfo.  */
rasInfo.RxOffset = 0;
rasInfo.RyOffset = 0;
rasInfo.Next = NULL;

InitVPort(&viewPort);           /*  Initialize the ViewPort.  */
view.ViewPort = &viewPort;      /*  Link the ViewPort into the View.  */
viewPort.RasInfo = &rasInfo;
viewPort.DWidth = WIDTH;
viewPort.DHeight = HEIGHT;

/* Set the display mode the old-fashioned way */
viewPort.Modes=HIRES | LACE;
```

```
if(GfxBase->LibNode.lib_Version >= 36)
{
    /* Make a ViewPortExtra and get ready to attach it */
    if( vpextra = GfxNew(VIEWPORT_EXTRA_TYPE) )
        {
        vcTags[1].ti_Data = (ULONG) vpextra;

        /* Initialize the DisplayClip field of the ViewPortExtra */
        if( GetDisplayInfoData( NULL , (UBYTE *) &dimquery ,
                                sizeof(dimquery) , DTAG_DIMS, modeID) )
            {
            vpextra->DisplayClip = dimquery.Nominal;

            /* Make a DisplayInfo and get ready to attach it */
            if( !(vcTags[2].ti_Data = (ULONG) FindDisplayInfo(modeID)) )
                fail("Could not get DisplayInfo\n");
             }
        else fail("Could not get DimensionInfo \n");
        }
    else fail("Could not get ViewPortExtra\n");

    /* This is for backwards compatibility with, for example,   */
    /* a 1.3 screen saver utility that looks at the Modes field */
    viewPort.Modes = (UWORD) (modeID & 0x0000ffff);
    }

/*  Initialize the ColorMap.  */
/*  2 planes deep, so 4 entries (2 raised to the #_planes power).  */
cm = GetColorMap(4L);
if(cm == NULL)
    fail("Could not get ColorMap\n");

if(GfxBase->LibNode.lib_Version >= 36)
    {
    /* Get ready to attach the ColorMap, Release 2-style */
    vcTags[0].ti_Data = (ULONG) &viewPort;

    /* Attach the color map and Release 2 extended structures */
    if( VideoControl(cm,vcTags) )
        fail("Could not attach extended structures\n");
    }
else
    /* Attach the ColorMap, old 1.3-style */
    viewPort.ColorMap = cm;

LoadRGB4(&viewPort, colortable, 4);  /* Change colors to those in colortable. */

MakeVPort( &view, &viewPort ); /* Construct preliminary Copper instruction list.    */

/* Merge preliminary lists into a real Copper list in the View structure. */
MrgCop( &view );

/* Clear the ViewPort */
for(depth=0; depth<DEPTH; depth++)
    {
    displaymem = (UBYTE *)bitMap.Planes[depth];
    BltClear(displaymem, (bitMap.BytesPerRow * bitMap.Rows), 1L);
    }

LoadView(&view);

/*  Now fill some boxes so that user can see something.         */
/*  Always draw into both planes to assure true colors.         */
for (box=1; box<=3; box++)  /* Three boxes; red, green and blue. */
    {
    for (depth=0; depth<DEPTH; depth++)        /*  Two planes.   */
        {
        displaymem = bitMap.Planes[depth] + boxoffsets[box-1];
        drawFilledBox(box, depth);
        }
    }

Delay(10L * TICKS_PER_SECOND);   /*  Pause for 10 seconds.               */
LoadView(oldview);               /*  Put back the old View.              */
WaitTOF();                       /*  Wait until the the View is being    */
                                 /*    rendered to free memory.          */
```

```
FreeCprList(view.LOFCprList);    /*  Deallocate the hardware Copper list  */
if(view.SHFCprList)              /*    created by MrgCop().  Since this   */
    FreeCprList(view.SHFCprList);/*    is interlace, also check for a     */
                                 /*    short frame copper list to free.   */
FreeVPortCopLists(&viewPort);    /*  Free all intermediate Copper lists   */
                                 /*    from created by MakeVPort().       */
cleanup(RETURN_OK);              /*  Success.                             */
}


/*
 * fail():  print the error string and call cleanup() to exit
 */
void fail(STRPTR errorstring)
{
printf(errorstring);
cleanup(RETURN_FAIL);
}

/*
 * cleanup():  free everything that was allocated.
 */
VOID cleanup(int returncode)
{
WORD depth;

/*  Free the color map created by GetColorMap().  */
if(cm) FreeColorMap(cm);

/* Free the ViewPortExtra created by GfxNew() */
if(vpextra) GfxFree(vpextra);

/*  Free the BitPlanes drawing area.  */
for(depth=0; depth<DEPTH; depth++)
    {
    if (bitMap.Planes[depth])
        FreeRaster(bitMap.Planes[depth], WIDTH, HEIGHT);
    }

/* Free the MonitorSpec created with OpenMonitor() */
if(monspec) CloseMonitor( monspec );

/* Free the ViewExtra created with GfxNew() */
if(vextra) GfxFree(vextra);

/* Close the graphics library */
CloseLibrary((struct Library *)GfxBase);

exit(returncode);
}


/*
 * drawFilledBox(): create a WIDTH/2 by HEIGHT/2 box of color
 *                  "fillcolor" into the given plane.
 */
VOID drawFilledBox(WORD fillcolor, WORD plane)
{
UBYTE value;
WORD boxHeight, boxWidth, width;

/*  Divide (WIDTH/2) by eight because each UBYTE that */
/* is written stuffs eight bits into the BitMap.      */
boxWidth = (WIDTH/2)/8;
boxHeight = HEIGHT/2;

value = ((fillcolor & (1 << plane)) != 0) ?  0xff : 0x00;

for( ; boxHeight; boxHeight--)
    {
    for(width=0 ; width < boxWidth; width++)
        *displaymem++ = value;

    displaymem += (bitMap.BytesPerRow - boxWidth);
    }
}
```

Exiting Gracefully

The preceding sample program provides a way of exiting gracefully with the **cleanup()** subroutine. This function returns to the memory manager all dynamically-allocated memory chunks. Notice the calls to **FreeRaster()** and **FreeColorMap()**. These calls correspond directly to the allocation calls **AllocRaster()** and **GetColorMap()** located in the body of the program. Now look at the calls within **cleanup()** to **FreeVPortCopLists()** and **FreeCprList()**. When you call **MakeVPort()**, the graphics system dynamically allocates some space to hold intermediate instructions from which a final Copper instruction list is created. When you call **MrgCop()**, these intermediate Copper lists are merged together into the final Copper list, which is then given to the hardware for interpretation. It is this list that provides the stable display on the screen, split into separate **ViewPorts** with their own colors and resolutions and so on.

When your program completes, you must see that it returns all of the memory resources that it used so that those memory areas are again available to the system for reassignment to other tasks. Therefore, if you use the routines **MakeVPort()** or **MrgCop()**, you must also arrange to use **FreeCprList()** (pointing to each of those lists in the **View** structure) and **FreeVPortCopLists()** (pointing to the **ViewPort** that is about to be deallocated). If your **View** is interlaced, you will also have to call **FreeCprList(&view.SHFCprList)** because an interlaced view has a separate Copper list for each of the two fields displayed. Do not confuse **FreeVPortCopLists()** with **FreeCprList()**. The former works on intermediate Copper lists for a specific **ViewPort**, the latter directly on a hardware Copper list from the **View**.

As a final caveat, notice that when you do free everything, the memory manager or other programs may immediately change the contents of the freed memory. Therefore, if the Copper is still executing an instruction stream (as a result of a previous **LoadView()**) when you free that memory, the display will malfunction. Once another **View** has been installed via **LoadView()**, do a **WaitTOF()** for the new **View** to begin displaying, and then you can begin freeing up your resources. **WaitTOF()** waits for the vertical blanking period to begin and all vertical blank interrupts to complete before returning to the caller. The routine **WaitBOVP()** (for "WaitBottomOfViewPort") *busy* waits until the vertical beam reaches the bottom of the specified **ViewPort** before returning to the caller. This means no other tasks run until this function returns.

MONITORS MODES, AND THE DISPLAY DATABASE

The Release 2 graphics library supports a variety of new video monitors, and new programmable video modes not available in older versions of the operating system. Inquiries about the availability of these modes, their dimensions and currently accessible options can be made through a database indexed by the same key information used to open Intuition screens. This design provides a good degree of compatibility with existing software, between differently equipped hardware platforms and for both static and dynamic data storage.

The Release 2 software may be running on A1000 computers which will not have ECS, on A500 computers which may not have the latest ECS upgrade, and on A2000 computers which generally have the latest ECS but may not have a multi-sync monitor currently attached. This means that there are compatibility issues to consider—what should happen when a required ECS or monitor resource is not available for the desired mode.

Here are the compatibility criteria, in a simplified fashion:

Requires Release 2, and ECS Chips only

SuperHires mode (35nS pixel resolutions). This allows for very high horizontal resolutions with the new ECS chip set and a standard NTSC or PAL monitor. (SuperHires has twice as much horizontal resolution as the old Hires mode.)

Requires Release 2, ECS Chips, and appropriate monitor

Productivity mode. This allows for flicker-free 640 x 480 color displays with the addition of a multi-sync or bi-sync 31 Khz monitor. (Productivity mode conforms, in a general way, to the VGA Mode 3 Standard 8514/A.)

Requires Release 2 (or the V35 of graphics.library under 1.3) and appropriate monitor only

A2024 Scan Conversion. This allows for a very high resolution grayscale display, typically 1008x800, suitable for desktop publishing or similar applications. A special video monitor is required (the monitor also supports the normal Amiga modes in greyscale).

Requires Release 2 but not ECS Chips or appropriate monitor

Display database inquiries. This allows for programmers to determine if the required resources are currently available for the requested mode.

In addition, there are fallback modes (which do not require Release 2) which resort to some reasonable display when a required resource is not available.

New Monitors

Currently, there are five possible monitor settings in the display database (more may be added in future releases):

default.monitor

Since the default system monitor must be capable of displaying an image no matter what chips are installed or what software revision is in ROM, the graphics.library default.monitor is defined as a 15 Khz monitor which can display NTSC in the U.S. or PAL in Europe.

ntsc.monitor

Since the ECS chip set allows for dynamic choice of standard scan rates, NTSC applications running on European machines may choose to be displayed on the ntsc.monitor to preserve the aspect ratio.

pal.monitor

Since the ECS chip set allows for dynamic choice of standard scan rates, PAL applications running on American machines may choose to be displayed on the pal.monitor to preserve the aspect ratio.

multisync.monitor

Programmably variable scan rates from 15 Khz through 31 Khz or more. Responds to signal timings to decide what scan rate to display. Required for Productivity (640 x 480 x 2 non-interlaced) display.

A2024.monitor

Scan converter monitor which provides 1008 x 800 x 2 (U.S.) or 1008 x 1024 x 2 (European) high-resolution, greyscale display. Does not require ECS. Does require Release 2 (or 1.3 V35) graphics library.

New Modes

In V1.3 and earlier versions of the OS, the mode for a display was determined by a 16 bit-value specified either in the **ViewPort.Modes** field (for displays set up with the graphics library) or in the **NewScreen.ViewModes** field (for displays set up with Intuition). Prior to Release 2, it was sufficient to indicate the mode of a display by setting bits in the **ViewPort.Modes** field. Furthermore, programs routinely made interpretations about a given display mode based on bit-by-bit testing of this 16-bit value.

Table 27-4: ViewPort Modes Used in 1.3

Bit	Name	1.3 ViewPort Modes	
15	HIRES	RP	
14	SPRITE	DC	
13	VPHIDE	DC	R = respected by 1.3
12	reserved	IP	I = ignored by 1.3
11	HAM	RP	D = dynamic
10	DUALPF	RP	C = cleared on write by 1.3 IFF writers
9	reserved	IP	P = preserved on write by 1.3 IFF writers
8	GENAUD	IC	
7	EHB	RP	
6	PF2PRI	RP	
5	reserved	IP	
4	reserved	IP	
3	reserved	IP	
2	LACE	RP	
1	GENVID	IC	
0	reserved	IP	

Considering all the possible new mode combinations and the need for future expansion, it is clear that the 16-bit mode specification used in 1.3 needs to be extended. Also, the specification of a mode needs to be separated from its interpretation. Furthermore, since modes can be grouped by the special monitor or physical device needed for the display, it is also beneficial to make provisions to support additional monitors and their modes in the future.

The approach taken in Release 2 is to introduce a new 32-bit display mode specifier called a ModeID. The upper half of this specifier is called the monitor part and the lower half is informally called the mode part. There is a correspondence between the monitor part and the monitor's operating modes (referred to as *virtual monitors* or **MonitorSpecs** after a system data structure).

For example, the A2024 monitor, PAL and NTSC are all different virtual monitors—the actual, physical monitor may be able to support more than one of these virtual types. Another new concept in Release 2 is the *default monitor*. The default monitor, represented by a zero value for the ModeID monitor part, may be either PAL or NTSC depending on a jumper on the motherboard.

Compatibility considerations—especially for IFF files and their CAMG chunk—have dictated very careful choices for the bit values which make up the mode part of the 32-bit ModeIDs. For example, the ModeIDs corresponding to the older, 1.3 display modes have been constructed out of a zero in the monitor part and the old 16-bit **ViewPort.Modes** bits in the lower part (after several extraneous bits such as SPRITES and VP_HIDE are cleared).

There are other such coincidences, but steps for compatibility with old programs notwithstanding, there is a new rule:

Programmers shall never interpret ModelDs on a bit-by-bit basis.

For example, if the HIRES bit is set it does not mean the display is 640 pixels wide because there can also be a doubling of the beam scan rate under Release 2. Programs should not attempt to interpret modes directly from the **ViewPort.Modes** field. The Release 2 graphics library provides a suitable substitute for this information through its new display database facility (explained below).

Likewise, under Release 2, the **Mode** of a **ViewPort** is no longer set directly. Instead it is set indirectly by associating the **ViewPort** with an abstract, 32-bit ModeID via the **VideoControl()** function.

These 32-bit ModeIDs have been carefully designed so that their lower 16 bits, when passed to graphics in the **ViewPort.Modes** field, provide some degree of compatibility between different systems. Older V1.3 programs will continue to work within the new scheme. (They will, however, not gain the benefits of the new modes and monitors available.)

Table 27-5: Extended ViewPort Modes Used in Release 2

Bit	Name	Release 2 ViewPort Modes	
15	MDBIT9	RP	
14	SPRITE	DC	
13	VPHIDE	DC	R = respected by Release
12	EXTEND	RP	I = ignored by Release 2
11	MDBIT8	RP	D = dynamic
10	MDBIT7	RP	C = cleared on write by Release 2 IFF writers
9	MDBIT6	RP	P = preserved on write by Release 2 IFF writers
8	reserved	IC	
7	MDBIT5	RP	
6	PF2PRI	RP	
5	MDBIT4	RP	
4	MDBIT3	RP	
3	MDBIT2	RP	
2	MDBIT1	RP	
1	reserved	IC	
0	MDBIT0	RP	

Refer to the example program, WBClone.c, at the end of this section for examples on opening Release 2 **ViewPorts** using the new ModeID specification.

Mode Specification, Screen Interface

Opening an Intuition screen in one of the new modes requires the specification of 32 bits of mode data. The **NewScreen.ViewModes** field is a UWORD (16 bits). Therefore, the new Release 2 function **OpenScreenTags()** must be used along with a SA_DisplayID tag which specifies the 32-bit ModeID. See the ''Intuition Screens'' chapter for more on this.

The new display modes also introduce some complexity for applications that want to support ''mode-sensitive'' processing. If a program wishes to open a screen in the highest resolution that a user has available, there are many more cases to handle under Release 2. Therefore, it will become increasingly important to algorithmically layout a screen for correct, functional and aesthetic operation. All the information needed to be mode-flexible is available through the display database functions (explained below).

Mode Specification, ViewPort Interface

When working directly with graphics, the interface is based on **View** and **ViewPort** structures, rather than on Intuition's **Screen** structure. As previously mentioned, new information must be associated with the **ViewPort** to specify the new Release 2 modes, and also with the **View** to specify what virtual monitor the whole **View** will be displayed on. There is also a lot of information to associate with a **ViewPort** regarding enhanced genlock capabilities.

This association of this new data with the **View** is made through a display database system which has been added to the Release 2 graphics library. All correctly written programs that allocate a **ColorMap** structure for a **ViewPort** use the **GetColorMap()** function to do it. Hence, in Release 2 the **ColorMap** structure is used as the general purpose black box extension of the **ViewPort** data.

To set or obtain the data in the extended structures, Release 2 provides a new function named **VideoControl()** which takes a **ColorMap** as an argument. This allows the setting and getting of the new extended display data. This mechanism is used to associate a **DisplayInfo** handle (*not* a ModeID) with a **ViewPort**. A **DisplayInfo** handle is an abstract link to the display database area associated with a particular ModeID. This handle is passed to the graphics database functions when getting or setting information about the mode. Using **VideoControl()**, a program can enable, disable, or obtain the state of a **ViewPort**'s **ColorMap**, mode, genlock and other features. The function uses a tag based interface and returns NULL if no error occurred.

```
error = BOOL VideoControl( struct ColorMap *cm, struct TagItem *tag );
```

The first argument is a pointer to a **ColorMap** structure as returned by the **GetColorMap()** function. The second argument is a pointer to an array of video control tag items, used to indicate whether information is being given or requested as well as to pass (or receive the information). The tags you can use with **VideoControl()** include the following:

VTAG_ATTACH_CM_GET (or _SET) is used to obtain the **ColorMap** structure from the indicated **ViewPort** or attach a given **ColorMap** to it.

VTAG_VIEWPORTEXTRA_GET (or _SET) is used to obtain the **ViewPortExtra** structure from the indicated **ColorMap** structure or attach a given **ViewPortExtra** to it. A **ViewPortExtra** structure is an extension of the **ViewPort** structure and should be allocated and freed with **GfxNew()** and **GfxFree()** and associated with the **ViewPort** with **VideoControl()**.

VTAG_NORMAL_DISP_GET (or _SET) is used to obtain or set the **DisplayInfo** structure for the standard or ''normal'' mode.

See *<graphics/videocontrol.h>* for a list of all the available tags. See the section on genlocking for information on using **VideoControl()** to interact with the Amiga's genlock capabilities. Note that the graphics library will modify the tag list passed to **VideoControl()**.

Coexisting Modes

Each display mode specifies (among other things) a pixel resolution and a monitor scan rate. Though the Amiga has the unique ability to change pixel resolutions on the fly, it is not possible to change the speed of a monitor beam in mid-frame. Therefore, if you set up a display of two or more **ViewPorts** in different display modes requiring different scan rates, at least one of the **ViewPorts** will be displayed with the wrong scan rate.

Such **ViewPorts** can be *coerced* into a different mode designed for the scan rate currently in effect. You can do this in a couple of ways, introducing or removing interlace to adjust the vertical dimension, and changing to faster or slower pixels (higher or lower resolution) for the horizontal dimension.

The disadvantage of introducing interlace is flicker. The disadvantage of increasing resolution is the lessening of the video bus bandwidth and possibly a reduction in the number of colors or palette resolution.

Under Intuition, the frontmost screen determines which of the conflicting modes will take precedence. With the graphics library, the **Modes** field of the **View** and its frontmost **ViewPort** or, in Release 2, the **MonitorSpec** of the **ViewExtra** determine the scan rate. For some monitors (such as the A2024), simultaneous display is excluded. This is a requirement only because the A2024 modes require very special and intricate display Copper list management.

ModeID Identifiers

The following definitions appear in the include file *<graphics/displayinfo.h>*. These values form the 32-bit ModeID which consists of a _MONITOR_ID in the upper word, and a _MODE_KEY in the lower word. Never interpret these bits directly. Instead use them with the display database to obtain the information you need about the display mode.

```
/* normal identifiers */

#define MONITOR_ID_MASK              0xFFFF1000

#define DEFAULT_MONITOR_ID           0x00000000
#define NTSC_MONITOR_ID              0x00011000
#define PAL_MONITOR_ID               0x00021000

/* the following 20 composite keys are for Modes on the default Monitor */
/* NTSC & PAL "flavors" of these particular keys may be made by OR'ing */
/* the NTSC or PAL MONITOR_ID with the desired MODE_KEY... */

#define LORES_KEY                    0x00000000
#define HIRES_KEY                    0x00008000
#define SUPER_KEY                    0x00008020
#define HAM_KEY                      0x00000800
#define LORESLACE_KEY                0x00000004
#define HIRESLACE_KEY                0x00008004
#define SUPERLACE_KEY                0x00008024
#define HAMLACE_KEY                  0x00000804
#define LORESDPF_KEY                 0x00000400
#define HIRESDPF_KEY                 0x00008400
#define SUPERDPF_KEY                 0x00008420
#define LORESLACEDPF_KEY             0x00000404
#define HIRESLACEDPF_KEY             0x00008404
#define SUPERLACEDPF_KEY             0x00008424
#define LORESDPF2_KEY                0x00000440
#define HIRESDPF2_KEY                0x00008440
#define SUPERDPF2_KEY                0x00008460
#define LORESLACEDPF2_KEY            0x00000444
#define HIRESLACEDPF2_KEY            0x00008444
#define SUPERLACEDPF2_KEY            0x00008464
#define EXTRAHALFBRITE_KEY           0x00000080
```

```
#define EXTRAHALFBRITELACE_KEY          0x00000084

/* vga identifiers */

#define VGA_MONITOR_ID                  0x00031000

#define VGAEXTRALORES_KEY               0x00031004
#define VGALORES_KEY                    0x00039004
#define VGAPRODUCT_KEY                  0x00039024
#define VGAHAM_KEY                      0x00031804
#define VGAEXTRALORESLACE_KEY           0x00031005
#define VGALORESLACE_KEY.               0x00039005
#define VGAPRODUCTLACE_KEY              0x00039025
#define VGAHAMLACE_KEY                  0x00031805
#define VGAEXTRALORESDPF_KEY            0x00031404
#define VGALORESDPF_KEY                 0x00039404
#define VGAPRODUCTDPF_KEY               0x00039424
#define VGAEXTRALORESLACEDPF_KEY        0x00031405
#define VGALORESLACEDPF_KEY             0x00039405
#define VGAPRODUCTLACEDPF_KEY           0x00039425
#define VGAEXTRALORESDPF2_KEY           0x00031444
#define VGALORESDPF2_KEY                0x00039444
#define VGAPRODUCTDPF2_KEY              0x00039464
#define VGAEXTRALORESLACEDPF2_KEY       0x00031445
#define VGALORESLACEDPF2_KEY            0x00039445
#define VGAPRODUCTLACEDPF2_KEY          0x00039465
#define VGAEXTRAHALFBRITE_KEY           0x00031084
#define VGAEXTRAHALFBRITELACE_KEY       0x00031085

/* a2024 identifiers */

#define A2024_MONITOR_ID                0x00041000

#define A2024TENHERTZ_KEY               0x00041000
#define A2024FIFTEENHERTZ_KEY           0x00049000

/* prototype identifiers */

#define PROTO_MONITOR_ID                0x00051000
```

The Display Database and the DisplayInfo Record

For each ModeID, the graphics library has a body of data that enables the set up of the display hardware and provides applications with information about the properties of the display mode.

The display information in the database is accessed by searching it for a record with a given ModeID. For performance reasons, a look-up function named **FindDisplayInfo()** is provided which, given a ModeID, will return a handle to the internal data record about the attributes of the display.

This handle is then used for queries to the display database and specification of display mode to the low-level graphics routines. It is never used as a pointer. The private data structure associated with a given ModeID is called a **DisplayInfo**. From the *<graphics/displayinfo.h>* include file:

```
/* the "public" handle to a DisplayInfo  */

 typedef APTR DisplayInfoHandle;
```

In order to obtain database information about an existing **ViewPort**, you must first gain reference to its 32-bit ModeID. A graphics function **GetVPModeID()** simplifies this operation:

```
modeID = ULONG GetVPModeID(struct ViewPort *vp )
```

The **vp** argument is pointer to a **ViewPort** structure. This function returns the normal display ModeID, if one is currently associated with this **ViewPort**. If no ModeID exists this function returns INVALID_ID.

Each new valid 32-bit ModelD is associated with data initialized by the graphics library at powerup. This data is accessed by obtaining a handle to it with the graphics function **FindDisplayInfo()**.

```
handle = DisplayInfoHandle FindDisplayInfo(ULONG modeID);
```

Given a 32-bit ModeID key (**modeID** in the prototype above) **FindDisplayInfo()** returns a handle to a valid **DisplayInfoRecord** found in the graphics database, or NULL. Using this handle, you can obtain information about this video mode, including its default dimensions, properties and whether it is currently available for use.

For instance, you can use a **DisplayInfoHandle** with the **GetDisplayInfoData()** function to look up the properties of a mode (see below). Or use the **DisplayInfoHandle** with **VideoControl()** and the VTAG_NORMAL_DISP_SET tag to set up a custom **ViewPort**.

Accessing the DisplayInfo

Basic information about a display can be obtained by calling the Release 2 graphics function **GetDisplayInfoData()**. You also call this function during the set up of a **ViewPort**.

```
result = ULONG GetDisplayInfoData( DisplayInfoHandle handle, UBYTE *buf,
                                   ULONG size, ULONG tagID, ULONG modeID )
```

Set the **handle** argument to the **DisplayInfoHandle** returned by a previous call to **FindDisplayInfo()**. This function will also accept a 32-bit ModeID directly as an argument. The **handle** argument should be set to NULL in that case.

The **buf** argument points to a destination buffer you have set up to hold the information about the properties of the display. The **size** argument gives the size of the buffer which depends on the type of inquiry you make.

The **tagID** argument specifies the type information you want to know about and may be set as follows:

DTAG_DISP Returns display properties and availability information (the buffer should be set to the size of a **DisplayInfo** structure).

DTAG_DIMS Returns default dimensions and overscan information (the buffer should be set to the size of a **DimensionInfo** structure).

DTAG_MNTR Returns monitor type, view position, scan rate, and compatibility (the buffer should be set to the size of a **MonitorInfo** structure).

DTAG_NAME Returns the user friendly name for this mode (the buffer should be set to the size of a **NameInfo** structure).

If the call succeeds, **result** is positive and reports the number of bytes actually transferred to the buffer. If the call fails (no information for the ModeID was available), **result** is zero.

Mode Availability

Even if the video monitor (NTSC, PAL, VGA, A2024) or ECS chips required to support a given mode are not available, there will be a **DisplayInfo** for all of the display modes. (This will not be the case for disk-based modes such as Euro36, Euro72, etc.)

Thus, the graphics library provides the **ModeNotAvailable()** function to determine whether a given mode is available, and if not, why not. Data corruption might cause the look-up function, **GetVPModeID()**, to fail even when it should not, so the careful programmer will always test the look-up function's return value.

```
error =  ULONG ModeNotAvailable( ULONG modeID )
```

The **modeID** argument is again a 32-bit ModeID as shown in *<graphics/displayinfo.h>*. This function returns an error code, indicating why this modeID is not available, or NULL if there is no known reason why this mode should not be there. The ULONG return values from this function are a proper superset of the **DisplayInfo.NotAvailable** field (defined in *<graphics/displayinfo.h>*).

The graphics library checks for the presence of the ECS chips at power up, but the monitor attached to the system cannot be detected and so must be specified by the user through a separate utility named AddMonitor.

Accessing the MonitorSpec

The **OpenMonitor()** function will locate and open the requested **MonitorSpec**. It is called with either the name of the monitor or a ModeID.

```
mspc = struct MonitorSpec *OpenMonitor(STRPTR name, ULONG modeID)
```

If the name argument is non-NULL, the **MonitorSpec** is chosen by name. If the name argument *is* NULL, the **MonitorSpec** is chosen by ModeID. If both the name and ModeID arguments are NULL, a pointer to the **MonitorSpec** for the *default* monitor is returned. **OpenMonitor()** returns either a pointer to a **MonitorSpec** structure, or NULL if the requested **MonitorSpec** could not be opened. The **CloseMonitor()** function relinquishes access to a **MonitorSpec** previously acquired with **OpenMonitor()**.

To set up a **View** in Release 2, a **ViewExtra** structure must also be created and attached to it. The **ViewExtra.Monitor** field must be initialized to the address of a valid **MonitorSpec** structure before the **View** is displayed. Use **OpenMonitor()** to initialize the **Monitor** field.

Mode Properties

Here is an example of how to query the properties of a given mode from a **DisplayInfoHandle**.

```
#include <graphics/displayinfo.h>

check_properties( handle )
DisplayInfoHandle handle;
{
    struct DisplayInfo queryinfo;

    /* fill in the displayinfo buffer with basic Mode display data */

    if(GetDisplayInfoData(handle, (UBYTE *)&queryinfo,sizeof(queryinfo),DTAG_DISP,NULL))
    {
```

```
        /* check for Properties of this Mode */

        if(queryinfo.PropertyFlags)
        {
            if(queryinfo.PropertyFlags & DIPF_IS_LACE)
                printf("mode is interlaced");
            if(queryinfo.PropertyFlags & DIPF_IS_DUALPF)
                printf("mode has dual playfields");
            if(queryinfo.PropertyFlags & DIPF_IS_PF2PRI)
                printf("mode has playfield two priority");
            if(queryinfo.PropertyFlags & DIPF_IS_HAM)
                printf("mode uses hold-and-modify");
            if(queryinfo.PropertyFlags & DIPF_IS_ECS)
                printf("mode requires the ECS chip set");
            if(queryinfo.PropertyFlags & DIPF_IS_PAL)
                printf("mode is naturally displayed on pal.monitor");
            if(queryinfo.PropertyFlags & DIPF_IS_SPRITES)
                printf("mode has sprites");
            if(queryinfo.PropertyFlags & DIPF_IS_GENLOCK)
                printf("mode is compatible with genlock displays");
            if(queryinfo.PropertyFlags & DIPF_IS_WB)
                printf("mode will support workbench displays");
            if(queryinfo.PropertyFlags & DIPF_IS_DRAGGABLE)
                printf("mode may be dragged to new positions");
            if(queryinfo.PropertyFlags & DIPF_IS_PANELLED)
                printf("mode is broken up for scan conversion");
            if(queryinfo.PropertyFlags & DIPF_IS_BEAMSYNC)
                printf("mode supports beam synchronization");
        }
    }
}
```

Nominal Values

Some of the display information is initialized in ROM for each mode such as recommended nominal (or default) dimensions. Even though this information is presumably static, it would still be a mistake to hardcode assumptions about these nominal values into your code.

Gathering information about the nominal dimensions of various modes is handled in a fashion similar to to the basic queries above. Here is an example of how to query the nominal dimensions of a given mode from a **DisplayInfoHandle**.

```
#include <graphics/displayinfo.h>

check_dimensions( handle )
DisplayInfoHandle handle;
{
    struct DimensionInfo query;

    /* fill the buffer with Mode dimension information */

    if(GetDisplayInfoData(handle, (UBYTE *)&query,sizeof(query),DTAG_DIMS,NULL)))
    {
        /* display Nominal dimensions of this Mode */

        printf("nominal width  = %ld",
                query.Nominal.MaxX - query.Nominal.MinX + 1);

        printf("nominal height = %ld",
                query.Nominal.MaxY - query.Nominal.MinY + 1);
    }
}
```

Preference Items

Some display information is changed in response to user Preference specification. Until further notice, this will be reserved as a system activity and use private interface methods.

One Preferences setting that may affect the display data is the user's preferred overscan limits to the monitor associated with this mode. Here is an example of how to query the overscan dimensions of a given mode from a **DisplayInfoHandle**.

```
#include <graphics/displayinfo.h>

check_overscan( handle )
DisplayInfoHandle handle;
{
    struct DimensionInfo query;

    /* fill the buffer with Mode dimension information */

    if(GetDisplayInfoData(handle, (UBYTE *)&query,sizeof(query),DTAG_DIMS,NULL)))
    {
        /* display standard overscan dimensions of this Mode */

        printf("overscan width  = %ld",
                query.StdOScan.MaxX - query.StdOScan.MinX + 1);

        printf("overscan height = %ld",
                query.StdOScan.MaxY - query.StdOScan.MinY + 1);
    }
}
```

Run-Time Name Binding of Mode Information

It is useful to associate common names with the various display modes. The Release 2 graphics library includes a provision for binding a name to a display mode so that it will be available via a query. This will be useful in the implementation of a standard screen-format requester. Note however that no names are bound initially since the bound names will take up RAM at all times. Instead defaults are used.

Bound names will override the defaults though, so that, until the screen-format requester is localized to a non-English language, the modes can be localized by binding foreign language names to them. Here is an example of how to query the run-time name binding of a given mode from a **DisplayInfoHandle**.

```
#include <graphics/displayinfo.h>

check_name_bound( handle )
DisplayInfoHandle handle;
{
    struct NameInfo query;

    /* fill the buffer with Mode dimension information */

    if(GetDisplayInfoData(handle, (UBYTE *)&query,sizeof(query),DTAG_NAME,NULL)))
    {
        printf("%s", query.Name);
    }
}
```

Relase 2 Custom ViewPort Example

The following program will create a display with the same attributes as the user's Workbench screen. It does this by first inquiring as to those attributes, duplicating them, and then creating a similar display.

```
/*********************************************************************/
/*                                                                 , */
/*      WBClone.c: To clone the Workbench using graphics calls       */
/*                                                                  */
/*      Compile : SAS/C 5.10a LC -b1 -cfist -L -v -y                */
/*                                                                  */
/*********************************************************************/

#include <exec/types.h>
#include <exec/exec.h>
#include <clib/exec_protos.h>
#include <intuition/screens.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <clib/intuition_protos.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/view.h>
#include <graphics/gfxnodes.h>
#include <graphics/videocontrol.h>
#include <clib/graphics_protos.h>

#include <stdio.h>
#include <stdlib.h>

#define INTUITIONNAME "intuition.library"

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }   /* really */
#endif

/*********************************************************************/
/*                          GLOBAL VARIABLES                        */
/*********************************************************************/

struct IntuitionBase *IntuitionBase = NULL ;
struct GfxBase *GfxBase = NULL ;

/*********************************************************************/
/*                                                                  */
/* VOID Error (char *String)                                        */
/*                                                                  */
/* Print string and exit                                            */
/*                                                                  */
/*********************************************************************/

VOID Error (char *String)
{
     VOID CloseAll (VOID) ;

     printf (String) ;

     CloseAll () ;
     exit(0) ;
}

/*********************************************************************/
/*                                                                  */
/* VOID Init ()                                                     */
/*                                                                  */
/* Opens all the required libraries allocates all memory, etc.      */
/*                                                                  */
/*********************************************************************/

VOID Init ( VOID )
{
```

```
    /* Open the intuition library.... */
    if ((IntuitionBase = (struct IntuitionBase *)OpenLibrary (INTUITIONNAME, 37L)) == NULL)
        Error ("Could not open the Intuition.library") ;

    /* Open the graphics library.... */
    if ((GfxBase = (struct GfxBase *)OpenLibrary (GRAPHICSNAME, 36L)) == NULL)
        Error ("Could not open the Graphics.library") ;
}

/*********************************************************************/
/*                                                                   */
/* VOID CloseAll ()                                                  */
/*                                                                   */
/* Closes and tidies up everything that was used.                    */
/*                                                                   */
/*********************************************************************/

VOID CloseAll ( VOID )
{
    /* Close everything in the reverse order in which they were opened */

    /* Close the Graphics Library */
    if (GfxBase)
        CloseLibrary ((struct Library *) GfxBase) ;

    /* Close the Intuition Library */
    if (IntuitionBase)
        CloseLibrary ((struct Library *) IntuitionBase) ;
}
/*********************************************************************/
/*                                                                   */
/* VOID DestroyView(struct View *view)                               */
/*                                                                   */
/* Close and free everything to do with the View                     */
/*                                                                   */
/*********************************************************************/

VOID DestroyView(struct View *view)
{
    struct ViewExtra *ve;

    if (view)
    {
        if (ve = (struct ViewExtra *)GfxLookUp(view))
        {
            if (ve->Monitor)
                CloseMonitor(ve->Monitor);

            GfxFree((struct ExtendedNode *)ve);
        }

        /* Free up the copper lists */
        if (view->LOFCprList)
            FreeCprList(view->LOFCprList);

        if (view->SHFCprList)
            FreeCprList(view->SHFCprList);

        FreeVec(view);
    }
}

/*********************************************************************/
/*                                                                   */
/* struct View *DupView(struct View *v, ULONG ModeID)                */
/*                                                                   */
/* Duplicate the View.                                               */
/*                                                                   */
/*********************************************************************/

struct View *DupView(struct View *v, ULONG ModeID)
{
    /* Allocate and init a View structure.  Also, get a ViewExtra
     * structure and attach the monitor type to the View.
     */
```

```
	struct View *view = NULL;
	struct ViewExtra *ve = NULL;
	struct MonitorSpec *mspc = NULL;

	if (view = AllocVec(sizeof(struct View), MEMF_PUBLIC | MEMF_CLEAR))
	{
		if (ve = GfxNew(VIEW_EXTRA_TYPE))
		{
			if (mspc = OpenMonitor(NULL, ModeID))
			{
				InitView(view);
				view->DyOffset = v->DyOffset;
				view->DxOffset = v->DxOffset;
				view->Modes = v->Modes;
				GfxAssociate(view, (struct ExtendedNode *)ve);
				ve->Monitor = mspc;
			}
			else printf("Could not open monitor\n");
		}
		else printf("Could not get ViewExtra\n");
	}
	else printf("Could not create View\n");

	if (view && ve && mspc)
		return(view);
	else
	{
		DestroyView(view);
		return(NULL);
	}
}


/***********************************************************************/
/*                                                                     */
/* VOID DestroyViewPort(struct ViewPort *vp)                           */
/*                                                                     */
/* Close and free everything to do with the ViewPort.                  */
/*                                                                     */
/***********************************************************************/

VOID DestroyViewPort(struct ViewPort *vp)
{
	if (vp)
	{
		/* Find the ViewPort's ColorMap. From that use VideoControl
		 *  to get the ViewPortExtra, and free it.
		 * Then free the ColorMap, and finally the ViewPort itself.
		 */
		struct ColorMap *cm = vp->ColorMap;
		struct TagItem ti[] =
		{
			{VTAG_VIEWPORTEXTRA_GET, NULL},     /* <-- This field will be filled in */
			{VTAG_END_CM, NULL}
		};

		if (cm)
		{
			if (VideoControl(cm, ti) == NULL)
				GfxFree((struct ExtendedNode *)ti[0].ti_Data);
			else
				printf("VideoControl error in DestroyViewPort()\n");

			FreeColorMap(cm);
		}
		else
		{
			printf("Could not free the ColorMap\n");
		}

		FreeVPortCopLists(vp);

		FreeVec(vp);
	}
}
```

```
/*********************************************************************/
/*                                                                   */
/* struct ViewPort *DupViewPort(struct ViewPort *vp, ULONG ModeID)   */
/*                                                                   */
/* Duplicate the ViewPort.                                           */
/*                                                                   */
/*********************************************************************/

struct ViewPort *DupViewPort(struct ViewPort *vp, ULONG ModeID)
{
      /* Allocate and initialise a ViewPort. Copy the ViewPort width and
       * heights, offsets, and modes values.  Allocate and initialize a
       * ColorMap.
       *
       * Also, allocate a ViewPortExtra, and copy the TextOScan values of the
       * ModeID from the database into the ViewPortExtra.
       */

      #define COLOURS 32
      struct ViewPort *Myvp;
      struct ViewPortExtra *vpe;
      struct ColorMap *cm;
      struct TagItem ti[] =               /* to attach everything */
      {
            {VTAG_ATTACH_CM_SET, NULL},   /* these NULLs will be replaced in the code */
            {VTAG_VIEWPORTEXTRA_SET, NULL},
            {VTAG_NORMAL_DISP_SET, NULL},
            {VTAG_END_CM, NULL}
      };
      struct DimensionInfo query = {0};
      UWORD colour;
      int c;
      ULONG gotinfo = NULL;

      if (Myvp = AllocVec(sizeof(struct ViewPort), MEMF_CLEAR | MEMF_PUBLIC))
      {
            if (vpe = (struct ViewPortExtra *)GfxNew(VIEWPORT_EXTRA_TYPE))
            {
                  if (cm = GetColorMap(32))
                  {
                        if (gotinfo = GetDisplayInfoData(NULL, (APTR)&query,
                                                           sizeof(query), DTAG_DIMS, ModeID))
                        {
                              InitVPort(Myvp);

                              /* duplicate the ViewPort structure */
                              Myvp->DWidth = vp->DWidth;
                              Myvp->DHeight = vp->DHeight;
                              Myvp->DxOffset = vp->DxOffset;
                              Myvp->DyOffset = vp->DyOffset;
                              Myvp->Modes = vp->Modes;
                              Myvp->SpritePriorities = vp->SpritePriorities;
                              Myvp->ExtendedModes = vp->ExtendedModes;

                              /* duplicate the Overscan values */
                              vpe->DisplayClip = query.TxtOScan;

                              /* attach everything together */
                              ti[0].ti_Data = (ULONG)Myvp;
                              ti[1].ti_Data = (ULONG)vpe;
                              ti[2].ti_Data = (ULONG)FindDisplayInfo(ModeID);
                              if (VideoControl(cm, ti) != NULL)
                              {
                                    printf("VideoControl error in CreateViewPort()\n");
                              }

                              /* copy the colours from the workbench */
                              for (c = 0; c < COLOURS; c++)
                              {
                                    if ((colour = GetRGB4(vp->ColorMap, c)) != -1)
                                    {
                                    SetRGB4CM(cm, c, (colour >> 8),
                                            ((colour >> 4) & 0xf), (colour & 0xf));
                                    }
                              }
                        }
```

```
                    else printf("Database error\n");
                }
                else printf("Could not get the ColorMap\n");
            }
            else printf("Could not get the ViewPortExtra\n");
        }
        else printf("Could not get the ViewPort\n");

        if (Myvp && vpe && cm && gotinfo)
            return(Myvp);
        else
        {
            DestroyViewPort(vp);
            return(NULL);
        }
}

/**********************************************************************************/
/*                                                                                */
/* VOID DestroyBitMap(struct BitMap *Mybm, SHORT width, SHORT height, SHORT depth) */
/*                                                                                */
/* Close and free everything to do with the BitMap                               */
/*                                                                                */
/**********************************************************************************/

VOID DestroyBitMap(struct BitMap *Mybm, SHORT width, SHORT height, SHORT depth)
{
        int i;

        if (Mybm)
        {
            for (i = 0; (i < depth); i++)
            {
                if (Mybm->Planes[i])
                    FreeRaster(Mybm->Planes[i], width, height);
            }
            FreeVec(Mybm);
        }
}

/**********************************************************************/
/*                                                                    */
/* struct BitMap *CreateBitMap(SHORT width, SHORT height, SHORT depth) */
/*                                                                    */
/* Create the BitMap.                                                 */
/*                                                                    */
/**********************************************************************/

struct BitMap *CreateBitMap(SHORT width, SHORT height, SHORT depth)
{
        /* Allocate a BitMap structure, initialise it, and allocate each plane. */

        struct BitMap *Mybm;
        PLANEPTR allocated = (PLANEPTR) 1;
        int i;

        if (Mybm = AllocVec(sizeof(struct BitMap), MEMF_CLEAR | MEMF_PUBLIC))
        {
            InitBitMap(Mybm, depth, width, height);
            for (i = 0; ((i < depth) && (allocated)); i++)
                allocated = (Mybm->Planes[i] = AllocRaster(width, height));

            if (allocated == NULL)
            {
                printf("Could not allocate all the planes\n");
                DestroyBitMap(Mybm, width, height, depth);
                Mybm = NULL;
            }
        }
        else printf("Could not get BitMap\n");

        return(Mybm);
}
```

```
/*****************************************************************************/
/*                                                                           */
/* VOID ShowView(struct View *view, struct ViewPort *vp, struct BitMap *bm,  */
/*                                          SHORT width, SHORT height)       */
/*                                                                           */
/* Assemble and display the View.                                            */
/*                                                                           */
/*****************************************************************************/

VOID ShowView(struct View *view, struct ViewPort *vp, struct BitMap *bm,
                                          SHORT width, SHORT height)
{
      /* Attach the BitMap to the ViewPort via a RasInfo.  Attach the ViewPort
       * to the View.  Clear the BitMap, and draw into it by attaching the BitMap
       * to a RastPort.  Then MakeVPort(), MrgCop() and LoadView().
       * Just wait for the user to press <RETURN> before returning.
       */

      struct RastPort *rp;
      struct RasInfo *ri;

      if (rp = AllocVec(sizeof(struct RastPort), MEMF_CLEAR | MEMF_PUBLIC))
      {
            if (ri = AllocVec(sizeof(struct RasInfo), MEMF_CLEAR | MEMF_PUBLIC))
            {
                  InitRastPort(rp);
                  ri->BitMap = rp->BitMap = bm;
                  vp->RasInfo = ri;
                  view->ViewPort = vp;

                  /* render */
                  SetRast(rp, 0);          /* clear the background */
                  SetAPen(rp, ((1 << bm->Depth) - 1));       /* use the last pen */
                  Move(rp, 0, 0);
                  Draw(rp, width, 0);
                  Draw(rp, width, height);
                  Draw(rp, 0, height);
                  Draw(rp, 0, 0);

                  /* display it */
                  MakeVPort(view, vp);
                  MrgCop(view);
                  LoadView(view);

                  getchar();

                  /* bring back the system */
                  RethinkDisplay();

                  FreeVec(ri);
            }
            else printf("Could not get RasInfo\n");

            FreeVec(rp);
      }
      else printf("Could not get RastPort\n");

}


/*********************************************************************/
/*                                                                   */
/* VOID main (int argc, char *argv[])                                */
/*                                                                   */
/* Clone the Workbench View using Graphics Library calls.            */
/*                                                                   */
/*********************************************************************/

VOID main (int argc, char *argv[])
{
      struct Screen *wb;
      struct View *Myview;
      struct ViewPort *Myvp;
      struct BitMap *Mybm;
      ULONG ModeID;
      ULONG IbaseLock;
```

```
    Init () ;          /* to open the libraries */

    /* To clone the Workbench using graphics calls involves duplicating
     * the Workbench ViewPort, ViewPort mode, and Intuition's View.
     * This also involves duplicating the DisplayClip for the overscan
     * value, the colours, and the View position.
     *
     * When this is all done, the View, ViewPort, ColorMap and BitMap
     * (and ViewPortExtra, ViewExtra and RasInfo) all have to be linked
     * together, and the copperlists made to create the display.
     *
     * This is not as difficult as it sounds (trust me!)
     */

    /* First, lock the Workbench screen, so no changes can be made to it
     * while we are duplicating it.
     */
    if (wb = LockPubScreen("Workbench"))
    {
        /* Find the Workbench's ModeID. This is a 32-bit number that
         * identifies the monitor type, and the display mode of that monitor.
         */
        ModeID = GetVPModeID(&wb->ViewPort);

        /* We need to duplicate Intuition's View structure, so lock IntuitionBase
         * to prevent the View changing under our feet.
         */
        IbaseLock = LockIBase(0);
        if (Myview = DupView(&IntuitionBase->ViewLord, ModeID))
        {
            /* The View has been cloned, so we don't need to keep it locked. */
            UnlockIBase(IbaseLock);

            /* Now duplicate the Workbench's ViewPort. Remember, we still have
             * the Workbench locked.
             */
            if (Myvp = DupViewPort(&wb->ViewPort, ModeID))
            {
                /* Create a BitMap to render into. This will be of the
                 * same dimensions as the Workbench.
                 */
                if (Mybm = CreateBitMap(wb->Width, wb->Height, wb->BitMap.Depth))
                {
                    /* Now we have everything copied, show something */
                    ShowView(Myview, Myvp, Mybm, wb->Width-1, wb->Height-1);

                    /* Now free up everything we have allocated */
                    DestroyBitMap(Mybm, wb->Width, wb->Height, wb->BitMap.Depth);
                }
                DestroyViewPort(Myvp);
            }
            DestroyView(Myview);
        }
        else
        {
            UnlockIBase(IbaseLock);
        }
        UnlockPubScreen(NULL, wb);
    }
    CloseAll () ;
}
```

Advanced Topics

This section covers advanced display topics such as dual-playfield mode, double-buffering, EHB mode and HAM mode.

CREATING A DUAL-PLAYFIELD DISPLAY

In dual-playfield mode, you have two separately controllable playfields. You specify dual-playfield mode in 1.3 by setting the DUALPF bit in the **ViewPort.Modes** field. In Release 2, you specify dual-playfield by using any ModeID that includes DPF in its name as listed in *<graphics/displayinfo.h>*.

In dual-playfield mode, you always define two **RasInfo** data structures. Each of these structures defines one of the playfields. There are five different ways you can configure a dual-playfield display, because there are five different distributions of the bitplanes which the system hardware allows.

Table 27-6: Bitplane Assignment in Dual-playfield Mode

Number of Bitplanes	Playfield 1 Depth	Playfield 2 Depth
2	1	1
3	2	1
4	2	2
5	3	2
6	3	3

Under 1.3 if PFBA is set in the **ViewPort.Modes** field, or, under Release 2, if the ModeID includes DPF2 in its name, then the playfield priorities are swapped and playfield 2 will be displayed in front of playfield 1. In this way, you can get more bitplanes in the background playfield than you have in the foreground playfield. The playfield priority affects only one **ViewPort** at a time. If you have multiple **ViewPorts** with dual-playfields, the playfield priority is set for each one individually.

Here's a summary of the steps you need to take to create a dual-playfield display:

- Allocate one **View** structure and one **ViewPort** structure.

- Allocate two **BitMap** structures. Allocate two **RasInfo** structures (linked together), each pointing to a separate **BitMap**. The two **RasInfo** structures are linked together as follows:

```
struct RasInfo playfield1, playfield2;

playfield1.Next = &playfield2;
playfield2.Next = NULL;
```

- Initialize each **BitMap** structure to describe one playfield, using one of the permissible bitplane distributions shown in the above table and allocate memory for the bitplanes themselves. Note that **BitMap** 1 and **BitMap** 2 need *not* be the same width and height.

- Initialize the **ViewPort** structure. In 1.3, specify dual-playfield mode by setting the DUALPF bit (and PFBA, if appropriate) in the **ViewPort.Modes** field. Under Release 2, specify dual-playfield mode by selecting a ModeID that includes DPF (or DPF2) in its name as listed in *<graphics/displayinfo.h>*. Set the **ViewPort.RasInfo** field to the address of the playfield 1 **RasInfo**.

- Set up the **ColorMap** information

- Call **MakeVPort()**, **MrgCop()** and **LoadView()** to display the newly created **ViewPort**.

For display purposes, each of the two **BitMaps** is assigned to a separate **ViewPort**. To *draw* separately into the **BitMaps**, you must also assign these **BitMaps** to two separate **RastPorts**. The section called "Initializing a RastPort Structure" shows you how to use a **RastPort** data structure to control your drawing routines.

CREATING A DOUBLE-BUFFERED DISPLAY

To produce smooth animation or similar effects, it is occasionally necessary to double-buffer your display. To prevent the user from seeing your graphics rendering while it is in progress, you will want to draw into one memory area while actually displaying a different area.

There are two methods of creating and displaying a double-buffered display. The simplest method is to create two complete **Views** and switch back and forth between them with **LoadView()** and **WaitTOF()**.

The second method consists of creating two separate display areas and two sets of pointers to those areas for a single **View**. This is more complicated but takes less memory.

- Allocate one **ViewPort** structure and one **View** structure.

- Allocate two **BitMap** structures and one **RasInfo** structure. Initialize each **BitMap** structure to describe one drawing area and allocate memory for the bitplanes themselves. Initialize the **RasInfo** structure, setting the **RasInfo.BitMap** field to the address of one of the two **BitMaps** you created.

- Call **MakeVPort()**, **MrgCop()** and **LoadView()**. When you call **MrgCop()**, the system uses the information you have provided to create a Copper instruction list for the Copper to execute. The system allocates memory for a long-frame (LOF) Copper list and, if this is an interlaced display, a short-frame (SHF) Copper list as well. The system places a pointer to the long-frame Copper list in **View.LOFCprList** and a pointer to a short-frame Copper list (if this is an interlaced display) in **View.SHFCprList**. The Copper instruction stream referenced by these pointers applies to the first **BitMap**.

- Save the values in **View.LOFCprList** and **View.SHFCprlist** and reset these fields to zero. Place a pointer to the second **BitMap** structure in the **RasInfo.BitMap** field. Next call **MakeVPort()** and **MrgCop()**.

- When you perform **MrgCop()** with the Copper instruction list fields of the **View** set to zero, the system automatically allocates and fills in a new list of instructions for the Copper. Now you have created two sets of instruction streams for the Copper, one that works with data in the first **BitMap** and the other that works with data in the second **BitMap**.

- You can save pointers to the second list of Copper instructions as well. Then, to perform the double-buffering, alternate between the two Copper lists. The code for the double-buffering loop would be as follows: call **WaitTOF()**, change the Copper instruction list pointers in the **View**, call **LoadView()** to show one of the **BitMaps** while drawing into the other **BitMap**, and repeat.

Remember that you will have to call **FreeCprList()** on both sets of Copper lists when you have finished.

EXTRA-HALF-BRITE MODE

In the Extra-Half-Brite mode you can create a single-playfield, low-resolution display with up to 64 colors, double the normal maximum of 32. This requires your **ViewPort** to be defined with six bitplanes. Under 1.3, you specify EHB mode by setting the EXTRA_HALFBRITE bit in the **ViewPort.Modes** field. Under Release 2, you specify EHB by selecting any ModeID which includes EXTRAHALFBRITE in its name as defined in the include file *<graphics/displainfo.h>*.

When setting up the color palette for an EHB display, you only specify values for registers 0 to 31. If you draw using color numbers 0 through 31, the pixel you draw will be the color specified in that particular system color register. If you draw using a color number from 32 to 63, then the color displayed will be half the intensity value of the corresponding color register from 0 to 31. For example, if color register 0 is set to 0xFFF (white), then color number 32 would be half this value or 0x777 (grey).

EHB mode uses all six bitplanes. The color register (0 through 31) is obtained from the bit combinations from planes 5 to 1, in that order of significance. Plane 6 is used to determine whether the full intensity (bit value 0) color or half-intensity (bit value 1) color is to be displayed.

HOLD-AND-MODIFY MODE

In hold-and-modify mode you can create a single-playfield, low-resolution display in which 4,096 different colors can be displayed simultaneously. This requires your **ViewPort** to be defined with six bitplanes. Under 1.3, you specify HAM mode by setting the HAM flag in the **ViewPort.Modes** field. Under Release 2, you specify HAM by selecting any ModeID which includes HAM in its name as defined in *<graphics/displayinfo.h>*.

When you draw into the **BitMap** associated with this **ViewPort**, you can choose colors in one of four different ways. If you draw using color numbers 0 to 15, the pixel you draw will appear in the color specified in that particular system color register. If you draw with any other color value (16 to 63) the color displayed depends on the color of the pixel that is to the immediate left of this pixel on the screen. To see how this works, consider how the bitplanes are used in HAM.

Hold-and-modify mode requires six bitplanes. Planes 5 and 6 are used to modify the way bits from planes 1 through 4 are treated, as follows:

- If the bit combination from planes 6 and 5 for any given pixel is 00, normal color selection procedure is followed. Thus, the bit combinations from planes 4 to 1, in that order of significance, are used to choose one of 16 color registers (registers 0 through 15).
- If the bit combination in planes 6 and 5 is 01, the color of the pixel immediately to the left of this pixel is duplicated and then modified. The bit combinations from planes 4 through 1 are used to replace the four bits representing the blue value of the preceding pixel color. (No color registers are changed.)
- If the bit combination in planes 6 and 5 is 10, then the color of the pixel immediately to the left of this pixel is duplicated and modified. The bit combinations from planes 4 through 1 are used to replace the four bits representing the red value of the preceding pixel color.
- If the bit combination in planes 6 and 5 is 11, then the color of the pixel immediately to the left of this pixel is duplicated and modified. The bit combinations from planes 4 through 1 are used to replace the four bits representing the green value of the preceding pixel color.

You can use just five bitplanes in HAM mode. In that case, the data for the sixth plane is automatically assumed to be 0. Note that for the first pixel in each line, hold-and-modify begins with the background color. The color choice does *not* carry over from the preceding line.

> *Note:* Since a typical hold-and-modify pixel only changes one of the three RGB color values at a time, color selection is limited. HAM mode does allow for the display of 4,096 colors simultaneously, but there are only 64 color options for any given pixel (not 4,096). The color of a pixel depends on the color of the preceding pixel.

Drawing Routines

Most of the graphics drawing routines require information about how the drawing is to take place. For this reason, most graphics drawing routines use a data structure called a **RastPort**, that contains pointers to the drawing area and drawing variables such as the current pen color and font to use. In general, you pass a pointer to your **RastPort** structure as an argument whenever you call a drawing function.

THE RASTPORT STRUCTURE

The **RastPort** data structure can be found in the include files *<graphics/rastport.h>* and *<graphics/rastport.i>*. It contains the following information:

```
struct RastPort
{
    struct  Layer  *Layer;
    struct  BitMap *BitMap;
    UWORD          *AreaPtrn;  /* Ptr to areafill pattern */
    struct  TmpRas *TmpRas;
    struct  AreaInfo *AreaInfo;
    struct  GelsInfo *GelsInfo;
    UBYTE   Mask;              /* Write mask for this raster */
    BYTE    FgPen;             /* Foreground pen for this raster */
    BYTE    BgPen;             /* Background pen  */
    BYTE    AOlPen;            /* Areafill outline pen */
    BYTE    DrawMode;          /* Drawing mode for fill, lines, and text */
    BYTE    AreaPtSz;          /* 2^n words for areafill pattern */
    BYTE    linpatcnt;         /* Current line drawing pattern preshift */
    BYTE    dummy;
    UWORD   Flags;             /* Miscellaneous control bits */
    UWORD   LinePtrn;          /* 16 bits for textured lines */
    WORD    cp_x, cp_y;        /* Current pen position */
    UBYTE   minterms[8];
    WORD    PenWidth;
    WORD    PenHeight;
    struct  TextFont *Font;    /* Current font address */
    UBYTE   AlgoStyle;         /* The algorithmically generated style */
    UBYTE   TxFlags;           /* Text specific flags */
    UWORD   TxHeight;          /* Text height */
    UWORD   TxWidth;           /* Text nominal width */
    UWORD   TxBaseline;        /* Text baseline */
    WORD    TxSpacing;         /* Text spacing (per character) */
    APTR    *RP_User;
    ULONG   longreserved[2];
#ifndef GFX_RASTPORT_1_2
    UWORD   wordreserved[7];   /* Used to be a node */
    UBYTE   reserved[8];       /* For future use */
#endif
};
```

The sections that follow explain each of the items in the **RastPort** structure is used.

Initializing a BitMap Structure

Associated with the **RastPort** is another data structure called a **BitMap** which contains a description of the organization of the data in the drawing area. This tells the graphics library where in memory the drawing area is located and how it is arranged. Before you can set up a **RastPort** for drawing you must first declare and initialize a **BitMap** structure, defining the characteristics of the drawing area, as shown in the following example. This was already shown in the section called "Forming a Basic Display," but it is repeated here because it relates to drawing as well as to display routines. (You need not necessarily use the same **BitMap** for both the drawing and the display, e.g., double-buffered displays.)

```
#define DEPTH 2        /* Two planes deep. */
#define WIDTH 320      /* Width in pixels. */
#define HEIGHT 200     /* Height in scanlines. */

struct BitMap bitMap;

/* Initialize the BitMap. */
InitBitMap(&bitMap, DEPTH, WIDTH, HEIGHT);
```

Initializing a RastPort Structure

Once you have a **BitMap** set up, you can declare and initialize the **RastPort** and then link the **BitMap** into it. Here is a sample initialization sequence:

```
struct BitMap bitMap = {0};
struct RastPort rastPort = {0};

/* Initialize the RastPort and link the BitMap to it. */
InitRastPort(&rastPort);
rastPort.BitMap = &bitMap;
```

> *Initialize, Then Link.* You cannot link the bitmap in until after the **RastPort** has been initialized.

RastPort Area-fill Information

Two structures in the **RastPort**—**AreaInfo** and **TmpRas**—define certain information for area filling operations. The **AreaInfo** pointer is initialized by a call to the routine **InitArea()**.

```
#define AREA_SIZE 200

register USHORT i;
WORD areaBuffer[AREA_SIZE];
struct AreaInfo areaInfo = {0};

/*  Clear areaBuffer before calling InitArea().  */
for (i=0; i<AREA_SIZE; i++)
    areaBuffer[i] = 0;

InitArea(&areaInfo, areaBuffer, (AREA_SIZE*2)/5);
```

The area buffer *must* start on a *word* boundary. That is why the sample declaration shows **areaBuffer** as composed of unsigned words (200), rather than unsigned bytes (400). It still reserves the same amount of space, but aligns the data space correctly.

To use area fill, you must first provide a work space in memory for the system to store the list of points that define your area. You must allow a storage space of 5 bytes per vertex. To create the areas in the work space, you use the functions **AreaMove()**, **AreaDraw()**, and **AreaEnd()**.

Typically, you prepare the **RastPort** for area-filling by following the steps in the code fragment above and then linking your **AreaInfo** into the **RastPort** like so:

```
rastPort->AreaInfo = &areaInfo;
```

In addition to the **AreaInfo** structure in the **RastPort**, you must also provide the system with some work space to build the object whose vertices you are going to define. This requires that you initialize a **TmpRas** structure, then point to that structure for your **RastPort** to use.

Here is a code fragment that builds and initializes a **TmpRas**. First the **TmpRas** structure is initialized (via **InitTmpRas()**) then it is linked into the **RastPort** structure.

> *Allocate Enough Space.* The area to which **TmpRas.RasPtr** points must be at least as large as the area (width times height) of the largest rectangular region you plan to fill. Typically, you allocate a space as large as a single bitplane (usually 320 by 200 bits for Lores mode, 640 by 200 for Hires, and 1280 by 200 for SuperHires).

When you use functions that dynamically allocate memory from the system, you must remember to return these memory blocks to the system before your program exits. See the description of **FreeRaster()** in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

RastPort Graphics Element Pointer

The graphics element pointer in the **RastPort** structure is called **GelsInfo**. If you are doing graphics animation using the GELS system, this pointer must refer to a properly initialized **GelsInfo** structure. See the chapter on "Graphics Sprites, Bobs and Animation" for more information.

RastPort Write Mask

The write mask is a **RastPort** variable that determines which of the bitplanes are currently writable. For most applications, this variable is set to all bits on. This means that all bitplanes defined in the **BitMap** are affected by a graphics writing operation. You can selectively disable one or more bitplanes by simply specifying a 0 bit in that specific position in the control byte. For example:

```
#include <graphics/gfxmacros.h>

SetWrMsk(&rastPort, 0xFB);    /* disable bitplane 2 */
```

A useful application for the **Mask** field is to set or clear plane 6 while in the Extra-Half-Brite display mode to create shadow effects. For example:

```
SetWrMsk(&rastPort, 0xE0);              /* Disable planes 1 through 5. */

SetAPen(&rastPort, 0);                  /* Clear the Extra-Half-Brite bit */
RectFill(&rastPort, 20, 20, 40, 30);    /*   in the old rectangle. */

SetAPen(&rastPort, 32);                 /* Set the Extra-Half-Brite bit */
RectFill(&rastPort, 30, 25, 50, 35);    /*   in the new rectangle. */

SetWrMsk(&rastPort, -1);                /* Re-enable all planes. */
```

RastPort Drawing Pens

The Amiga has three different drawing "pens" associated with the graphics drawing routines. These are:

- **FgPen**—the foreground or primary drawing pen. For historical reasons, it is also called the A-Pen.
- **BgPen**—the background or secondary drawing pen. For historical reasons, it is also called the B-Pen.
- **AOlPen**—the area outline pen. For historical reasons, it is also called the O-Pen.

A drawing pen variable in the **RastPort** contains the current value (range 0-255) for a particular color choice. This value represents a color register number whose contents are to be used in rendering a particular type of image. The effect of the pen value is dependent upon the drawing mode and can be influenced by the pattern variables and the write mask as described below. Always use the system calls (e.g. **SetAPen()**) to set the different pens, never store values directly into the pen fields of the **RastPort**.

> *Colors Repeat Beyond 31.* The Amiga 500/2000/3000 (with original chips or ECS) contains only 32 color registers. Any range beyond that repeats the colors in 0-31. For example, pen numbers 32-63 refer to the colors in registers 0-31 (except when you are using Extra-Half-Brite mode).

The graphics library drawing routines support **BitMaps** up to eight planes deep allowing for future expansion of the Amiga hardware.

The color in **FgPen** is used as the primary drawing color for rendering lines and areas. This pen is used when the drawing mode is JAM1 (see the next section for drawing modes). JAM1 specifies that only one color is to be "jammed" into the drawing area.

You establish the color for **FgPen** using the statement:

```
SetAPen(&rastPort, newcolor);
```

The color in **BgPen** is used as the secondary drawing color for rendering lines and areas. If you specify that the drawing mode is JAM2 (jamming two colors) and a pattern is being drawn, the primary drawing color (**FgPen**) is used where there are 1s in the pattern. The secondary drawing color (**BgPen**) is used where there are 0s in the pattern.

You establish the drawing color for **BgPen** using the statement:

```
SetBPen(&rastPort, newcolor);
```

The area outline pen **AOlPen** is used in two applications: area fill and flood fill. (See "Area Fill Operations" below.) In area fill, you can specify that an area, once filled, can be outlined in this **AOlPen** color. In flood fill (in one of its operating modes) you can fill until the flood-filler hits a pixel of the color specified in this pen variable.

You establish the drawing color for **AOlPen** using the statement:

```
SetOPen(&rastPort, newcolor);
```

RastPort Drawing Modes

Four drawing modes may be specified in the **RastPort.DrawMode** field:

JAM1 Whenever you execute a graphics drawing command, one color is jammed into the target drawing area. You use only the primary drawing pen color, and for each pixel drawn, you *replace* the color at that location with the **FgPen** color.

JAM2 Whenever you execute a graphics drawing command, two colors are jammed into the target drawing area. This mode tells the system that the pattern variables (both line pattern and area pattern—see the next section) are to be used for the drawing. Wherever there is a 1 bit in the pattern variable, the **FgPen** color replaces the color of the pixel at the drawing position. Wherever there is a 0 bit in the pattern variable, the **BgPen** color is used.

COMPLEMENT
For each 1 bit in the pattern, the corresponding bit in the target area is complemented—that is, its state is reversed. As with all other drawing modes, the write mask can be used to protect specific bitplanes from being modified. Complement mode is often used for drawing and then erasing lines.

INVERSVID
This is the drawing mode used primarily for text. If the drawing mode is (JAM1 | INVERSVID), the text appears as a transparent letter surrounded by the **FgPen** color. If the drawing mode is (JAM2|INVERSVID), the text appears as in (JAM1|INVERSVID) except that the **BgPen** color is used to draw the text character itself. In this mode, the roles of **FgPen** and **BgPen** are effectively reversed.

You set the drawing modes using the statement:

```
SetDrMd(&rastPort, newmode);
```

Set the **newmode** argument to one of the four drawing modes listed above.

RastPort Line and Area Drawing Patterns

The **RastPort** data structure provides two different pattern variables that it uses during the various drawing functions: a line pattern and an area pattern. The line pattern is 16 bits wide and is applied to all lines. When you initialize a **RastPort**, this line pattern value is set to all 1s (hex FFFF), so that solid lines are drawn. You can also set this pattern to other values to draw dotted lines if you wish. For example, you can establish a dotted line pattern with the graphics macro **SetDrPt()**:

```
SetDrPt(&rastPort, 0xCCCC);
```

The second argument is a bit-pattern, 1100110011001100, to be applied to all lines drawn. If you draw multiple, connected lines, the pattern cleanly connects all the points.

The area pattern is also 16 bits wide and its height is some power of two. This means that you can define patterns in heights of 1, 2, 4, 8, 16, and so on. To tell the system how large a pattern you are providing, use the graphics macro **SetAfPt()**:

```
SetAfPt(&rastPort, &areaPattern, power_of_two);
```

The **&areaPattern** argument is the address of the first word of the area pattern and **power_of_two** specifies how many words are in the pattern. For example:

```
USHORT ditherData[] =
{
    0x5555, 0xAAAA
};

SetAfPt(&rastPort, ditherData, 1);
```

This example produces a small cross-hatch pattern, useful for shading. Because **power_of_two** is set to 1, the pattern height is 2 to the 1st, or 2 rows high.

To clear the area fill pattern, use:

```
SetAfPt(&rastPort, NULL, 0);
```

> *Pattern Positioning.* The pattern is always positioned with respect to the upper left corner of the **RastPort** drawing area (the 0,0 coordinate). If you draw two rectangles whose edges are adjacent, the pattern will be continuous across the rectangle boundaries.

The last example given produces a two-color pattern with one color where there are 1s and the other color where there are 0s in the pattern. A special mode allows you to develop a pattern having up to 256 colors. To create this effect, specify **power_of_two** as a negative value instead of a positive value. For instance, the following initialization establishes an 8-color checkerboard pattern where each square in the checkerboard has a different color.

```
USHORT areaPattern[3][8] =
{
/* plane 0 pattern */
    {
            0x0000, 0x0000,
            0xffff, 0xffff,
            0x0000, 0x0000,
            0xffff, 0xffff
    },
/* plane 1 pattern */
    {
            0x0000, 0x0000,
            0x0000, 0x0000,
            0xffff, 0xffff,
            0xffff, 0xffff
    },
/* plane 2 pattern */
    {
            0xff00, 0xff00,
            0xff00, 0xff00,
            0xff00, 0xff00,
            0xff00, 0xff00
    }
};

SetAfPt(&rastPort, &areaPattern, -3);

/* when doing this, it is best to set three other parameters as follows: */
SetAPen(&rastPort, -1);
SetBPen(&rastPort, 0);
SetDrMd(&rastPort, JAM2);
```

If you use this multicolored pattern mode, you must provide as many planes of pattern data as there are planes in your **BitMap**.

RastPort Pen Position and Size

The graphics drawing routines keep the current position of the drawing pen in the **RastPort** fields **cp_x** and **cp_y**, for the horizontal and vertical positions, respectively. The coordinate location 0,0 is in the upper left corner of the drawing area. The x value increases proceeding to the right; the y value increases proceeding toward the bottom of the drawing area.

The variables **RastPort.PenWidth** and **RastPort.PenHeight** are not currently implemented. These fields should not be read or written by applications.

Text Attributes

Text attributes and font information are stored in the **RastPort** fields **Font, AlgoStyle, TxFlags, TxHeight, TxWidth, TxBaseline** and **TxSpacing**. These are normally set by calls to the graphics font routines which are covered separately in the chapter on "Graphics Library and Text."

USING THE GRAPHICS DRAWING ROUTINES

This section shows you how to use the Amiga drawing routines. All of these routines work either on their own or along with the windowing system and layers library. For details about using the layers and windows, see the chapters on "Layers Library" and "Intuition Windows".

> *Use WaitBlit().* The graphics library rendering and data movement routines generally wait to get access to the blitter, start their blit, and then exit. Therefore, you must **WaitBlit()** after a graphics rendering or data movement call if you intend to immediately deallocate, examine, or perform order-dependent processor operations on the memory used in the call.

As you read this section, keep in mind that to use the drawing routines, you need to pass them a pointer to a **RastPort**. You can define the **RastPort** directly, as shown in the sample program segments in preceding sections, or you can get a **RastPort** from your **Window** structure using code like the following:

```
struct Window *window;
struct RastPort *rastPort;

window = OpenWindow(&newWindow);    /*  You could use OpenWindowTags() instead. */
if (window)
    rastPort = window->RPort;
```

You can also get the **RastPort** from the **Layer** structure, if you are not using Intuition.

Drawing Individual Pixels

You can set a specific pixel to a desired color by using a statement like this:

```
SHORT x, y;
LONG result;
result = WritePixel(&rastPort, x, y);
```

WritePixel() uses the primary drawing pen and changes the pixel at that x,y position to the desired color if the x,y coordinate falls within the boundaries of the **RastPort**. A value of 0 is returned if the write was successful; a value of -1 is returned if x,y was outside the range of the **RastPort**.

Reading Individual Pixels

You can determine the color of a specific pixel with a statement like this:

```
SHORT x, y;
LONG result;
result = ReadPixel(&rastPort, x, y);
```

ReadPixel() returns the value of the pixel color selector at the specified x,y location. If the coordinates you specify are outside the range of your **RastPort**, this function returns a value of -1.

Drawing Ellipses and Circles

Two functions are associated with drawing ellipses: **DrawCircle()** and **DrawEllipse()**. **DrawCircle()**, a macro that calls **DrawEllipse()**, will draw a circle from the specified center point using the specified radius. This function is executed by the statement:

```
DrawCircle(&rastPort, center_x, center_y, radius);
```

Similarly, **DrawEllipse()** draws an ellipse with the specified radii from the specified center point:

```
DrawEllipse(&rastPort, center_x, center_y, horiz_r, vert_r);
```

Neither function performs clipping on a non-layered **RastPort**.

Drawing Lines

Two functions are associated with line drawing: **Move()** and **Draw()**. **Move()** simply moves the cursor to a new position. It is like picking up a drawing pen and placing it at a new location. This function is executed by the statement:

```
Move(&rastPort, x, y);
```

Draw() draws a line from the current x,y position to a new x,y position specified in the statement itself. The drawing pen is left at the new position. This is done by the statement:

```
Draw(&rastPort, x, y);
```

Draw() uses the pen color specified for **FgPen**. Here is a sample sequence that draws a line from location (0,0) to (100,50).

```
SetAPen(&rastPort, COLOR1);    /* Set A pen color. */
Move(&rastPort, 0, 0);    /* Move to this location. */
Draw(&rastPort, 100,50);    /* Draw to a this location. */
```

> *Caution:* If you attempt to draw a line outside the bounds of the **BitMap**, using the basic initialized **RastPort**, you may crash the system. You must either do your own software clipping to assure that the line is in range, or use the layers library. Software clipping means that you need to determine if the line will fall outside your **BitMap** *before* you draw it, and render only the part which falls inside the **BitMap**.

Drawing Patterned Lines

To turn the example above into a patterned line draw, simply set a drawing pattern, such as:

```
SetDrPt(&rastPort, 0xAAAA);
```

Now all lines drawn appear as dotted lines (0xAAAA = 1010101010101010 in binary). To resume drawing solid lines, execute the statement:

```
SetDrPt(&rastPort, ~0);
```

Because ~0 is defined as all bits on (11...11) in binary.

Drawing Multiple Lines with a Single Command

You can use multiple **Draw()** statements to draw connected line figures. If the shapes are all definable as interconnected, continuous lines, you can use a simpler function, called **PolyDraw()**. **PolyDraw()** takes a set of line endpoints and draws a shape using these points. You call **PolyDraw()** with the statement:

```
PolyDraw(&rastPort, count, arraypointer);
```

PolyDraw() reads the array of points and draws a line from the first pair of coordinates to the second, then a connecting line to each succeeding pair in the array until **count** points have been connected. This function uses the current drawing mode, pens, line pattern, and write mask specified in the target **RastPort**; for example, this fragment draws a rectangle, using the five defined pairs of x,y coordinates.

```
SHORT linearray[] =
    {
     3, 3,
    15, 3,
    15,15,
     3,15,
     3, 3
    };

PolyDraw(&rastPort, 5, linearray);
```

Area-fill Operations

Assuming that you have properly initialized your **RastPort** structure to include a properly initialized **AreaInfo**, you can perform area fill by using the functions described in this section.

AreaMove() tells the system to begin a new polygon, closing off any other polygon that may already be in process by connecting the end-point of the previous polygon to its starting point. **AreaMove()** is executed with the statement:

```
LONG result;
result = AreaMove(&rastPort, x, y);
```

AreaMove() returns 0 if successful, -1 if there was no more space left in the vector list. **AreaDraw()** tells the system to add a new vertex to a list that it is building. No drawing takes place until **AreaEnd()** is executed. **AreaDraw** is executed with the statement:

```
LONG result;
result = AreaDraw(&rastPort, x, y);
```

AreaDraw() returns 0 if successful, -1 if there was no more space left in the vector list. **AreaEnd()** tells the system to draw all of the defined shapes and fill them. When this function is executed, it obeys the drawing mode and uses the line pattern and area pattern specified in your **RastPort** to render the objects you have defined.

To fill an area, you do not have to **AreaDraw()** back to the first point before calling **AreaEnd()**. **AreaEnd()** automatically closes the polygon. **AreaEnd()** is executed with the following statement:

```
LONG result;
result = AreaEnd(&rastPort);
```

AreaEnd() returns 0 if successful, -1 if there was an error. To turn off the outline function, you have to set the **RastPort Flags** variable back to 0 with BNDRYOFF():

```
#include "graphics/gfxmacros.h"

BNDRYOFF(&rastPort);
```

Otherwise, every subsequent area-fill or rectangle-fill operation will outline their rendering with the outline pen (**AOlPen**).

Ellipse and Circle-fill Operations

Two functions are associated with drawing filled ellipses: **AreaCircle()** and **AreaEllipse()**. **AreaCircle()** (a macro that calls **AreaEllipse()**) will draw a circle from the specified center point using the specified radius. This function is executed by the statement:

```
AreaCircle(&rastPort, center_x, center_y, radius);
```

Similarly, **AreaEllipse()** draws a filled ellipse with the specified radii from the specified center point:

```
AreaEllipse(&rastPort, center_x, center_y, horiz_r, vert_r);
```

Outlining with **SetOPen()** is not currently supported by the **AreaCircle()** and **AreaEllipse()** routines.

> *Caution:* If you attempt to fill an area outside the bounds of the **BitMap**, using the basic initialized **RastPort**, it may crash the system. You must either do your own software clipping to assure that the area is in range, or use the layers library.

Flood-fill Operations

Flood fill is a technique for filling an arbitrary shape with a color. The Amiga flood-fill routines can use a plain color or do the fill using a combination of the drawing mode, **FgPen**, **BgPen** and the area pattern.

Flood-fill requires a **TmpRas** structure at least as large as the **RastPort** in which the flood-fill will be done. This is to ensure that even if the flood-filling operation ''leaks'', it will not flow outside the **TmpRas** and corrupt another task's memory.

You use the **Flood()** routine for flood fill. The syntax for this routine is as follows:

```
Flood(&rastPort, mode, x, y);
```

The **rastPort** argument specifies the **RastPort** you want to draw into. The **x** and **y** arguments specify the starting coordinate within the **BitMap**. The **mode** argument tells how to do the fill. There are two different modes for flood fill:

Outline Mode

In outline mode, you specify an x,y coordinate, and from that point the system searches outward in all directions for a pixel whose color is the same as that specified in the area outline pen (**AOlPen**). All horizontally or vertically adjacent pixels *not* of that color are filled with a colored pattern or plain color. The fill stops at the outline color. Outline mode is selected when the **mode** argument to **Flood()** is set to a 0.

Color Mode

In color mode, you specify an x,y coordinate, and whatever pixel color is found at that position defines the area to be filled. The system searches for all horizontally or vertically adjacent pixels whose color is the same as this one and replaces them with the colored pattern or plain color. Color mode is selected when the **mode** argument ot **Flood()** is set to a one.

The following sample program fragment creates and then flood-fills a triangular region. The overall effect is exactly the same as shown in the preceding area-fill example above, except that flood-fill is slightly slower than area-fill. Mode 0 (fill to a pixel that has the color of the outline pen) is used in the example.

```
BYTE oldAPen;
UWORD oldDrPt;
struct RastPort *rastPort = Window->RPort;

/* Save the old values of the foreground pen and draw pattern. */
oldAPen = rastPort->FgPen;
oldDrPt = rastPort->LinePtrn;

/* Use AreaOutline pen color for foreground pen. */
SetAPen(rastPort, rastPort->AOlPen);
SetDrPt(rastPort, ~0);          /* Insure a solid draw pattern. */

Move(rastPort, 0, 0);           /* Using mode 0 to create a triangular shape */
Draw(rastPort, 0, 100);
Draw(rastPort, 100, 100);
Draw(rastPort, 0, 0);           /* close it */

SetAPen(rastPort, oldAPen);     /* Restore original foreground pen. */
Flood(rastPort, 0, 10, 50);     /* Start Flood() inside triangle. */

SetDrPt(rastPort, oldDrPt);     /* Restore original draw mode. */
```

This example saves the current **FgPen** value and draws the shape in the same color as **AOlPen**. Then **FgPen** is restored to its original color so that **FgPen**, **BgPen**, **DrawMode**, and **AreaPtrn** can be used to define the fill within the outline.

Rectangle-fill Operations

The final fill function, **RectFill()**, is for filling rectangular areas. The form of this function follows:

```
RectFill(&rastPort, xmin, ymin, xmax, ymax);
```

As usual, the **rastPort** argument specifies the **RastPort** you want to draw into. The **xmin** and **ymin** arguments specify the upper left corner of the rectangle to be filled. The **xmax** and **ymax** arguments specify the lower right corner of the rectangle to be filled. Note that the variable **xmax** must be equal to or greater than **xmin**, and **ymax** must be equal to or greater than **ymin**.

Rectangle-fill uses **FgPen**, **BgPen**, **AOlPen**, **DrawMode**, **AreaPtrn** and **Mask** to fill the area you specify. Remember that the fill can be multicolored as well as single- or two-colored. When the **DrawMode** is COMPLEMENT, it complements all bit planes, rather than only those planes in which the foreground is non-zero.

PERFORMING DATA MOVE OPERATIONS

The graphics library includes several routines that use the hardware blitter to handle the rectangularly organized data that you work with when doing raster-based graphics. These blitter routines do the following:

- Clear an entire segment of memory
- Set a raster to a specific color
- Scroll a subrectangle of a raster
- Draw a pattern ''through a stencil"
- Extract a pattern from a bit-packed array and draw it into a raster
- Copy rectangular regions from one bitmap to another
- Control and utilize the hardware-based data mover, the blitter

The following sections cover these routines in detail.

> *WARNING:* The graphics library rendering and data movement routines generally wait to get access to the blitter, start their blit, and then exit without waiting for the blit to finish. Therefore, you must **WaitBlit()** after a graphics rendering or data movement call if you intend to immediately deallocate, examine, or perform order-dependent processor operations on the memory used in the call.

Clearing a Memory Area

For memory that is accessible to the blitter (that is, internal Chip memory), the most efficient way to clear a range of memory is to use the blitter. You use the blitter to clear a block of memory with the statement:

```
BltClear(memblock, bytecount, flags);
```

The **memblock** argument is a pointer to the location of the first byte to be cleared and **bytecount** is the number of bytes to set to zero. In general the **flags** variable should be set to one to wait for the blitter operation to complete. Refer to the *Amiga ROM Kernel Manual: Includes and Autodocs* for other details about the **flag** argument.

Setting a Whole Raster to a Color

You can preset a whole raster to a single color by using the function **SetRast()**. A call to this function takes the following form:

```
SetRast(&rastPort, pen);
```

As always, the **&rastPort** is a pointer to the **RastPort** you wish to use. Set the **pen** argument to the color register you want to fill the **RastPort** with.

Scrolling a Sub-rectangle of a Raster

You can scroll a sub-rectangle of a raster in any direction—up, down, left, right, or diagonally. To perform a scroll, you use the **ScrollRaster()** routine and specify a **dx** and **dy** (delta-x, delta-y) by which the rectangle image should be moved relative to the (0,0) location.

As a result of this operation, the data within the rectangle will become physically smaller by the size of delta-x and delta-y, and the area vacated by the data when it has been cropped and moved is filled with the background color (color in **BgPen**). **ScrollRaster()** is affected by the **Mask** setting.

Here is the syntax of the **ScrollRaster()** function:

```
ScrollRaster(&rastPort, dx, dy, xmin, ymin, xmax, ymax);
```

The **&rastPort** argument is a pointer to a **RastPort**. The **dx** and **dy** arguments are the distances (positive, 0, or negative) to move the rectangle. The outer bounds of the sub-rectangle are defined by the **xmin**, **xmax**, **ymin** and **ymax** arguments.

Here are some examples that scroll a sub-rectangle:

```
/* scroll up 2 */
ScrollRaster(&rastPort, 0, 2, 10, 10, 50, 50);

/* scroll right 1 */
ScrollRaster(&rastPort, -1, 0, 10, 10, 50, 50);
```

When scrolling a Simple Refresh window (or other layered **RastPort**), **ScrollRaster()** scrolls the appropriate existing damage region. Refer to the "Intuition Windows" chapter for an explanation of Simple Refresh windows and damage regions.

When scrolling a SuperBitMap window **ScrollRaster()** requires a properly initialized **TmpRas**. The **TmpRas** must be initialized to the size of one bitplane with a width and height the same as the SuperBitMap, using the technique described in the "Area-Fill Information" section above.

If you are using a SuperBitMap **Layer**, it is possible that the information in the **BitMap** is not fully reflected in the layer and vice-versa. Two graphics calls, **CopySBitMap()** and **SyncSBitMap()**, remedy these situations. Again, refer to the "Intuition Windows" chapter for more on this.

Drawing through a Stencil

The routine **BltPattern()** allows you to change only a very selective portion of a drawing area. Basically, this routine lets you define a rectangular region to be affected by a drawing operation and a mask of the same size that further defines which pixels within the rectangle will be affected.

The figure below shows an example of what you can do with **BltPattern()**. The 0 bits are represented by blank rectangles; the 1 bits by filled-in rectangles.

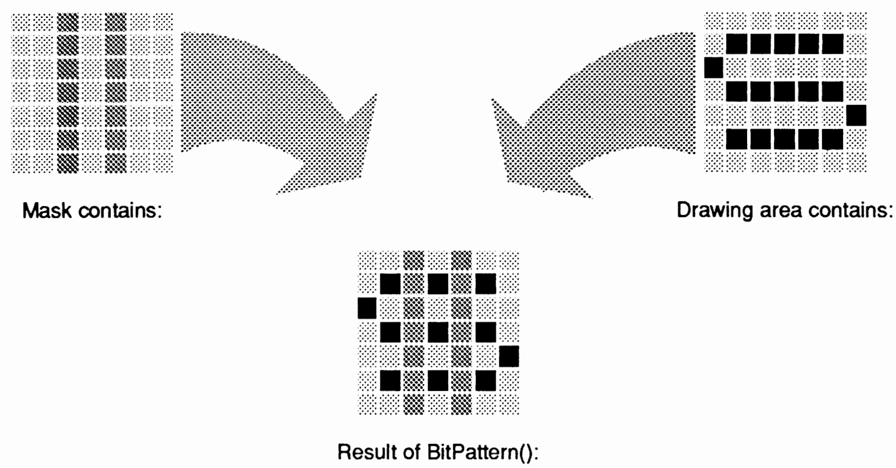


Figure 27-17: Example of Drawing Through a Stencil

In the resulting drawing, the lighter squares show where the target drawing area has been affected. Exactly *what* goes into the drawing area when the mask has 1's is determined by your **RastPort**'s **FgPen**, **BgPen**, **DrawMode** and **AreaPtrn** fields.

You call **BltPattern()** with:

```
BltPattern(&rastport, mask, xl, yl, maxx, maxy, bytecnt)
```

The **&rastport** argument specifies the **RastPort** to use. The operation will be confined to a rectangular area within the **RastPort** specified by **xl** and **yl** (upper right corner of the rectangle) and **maxx** and **maxy** (lower right corner of the rectangle).

The **mask** is a pointer to the mask to use. This can be NULL, in which case a simple rectangular region is modified. Or it can be set to the address of a byte pattern which allows any arbitrary shape within the rectangle to be defined. The **bytecount** is the number of bytes per row for the mask (it *must* be an even number of bytes).

The **mask** parameter is a rectangularly organized, contiguously stored pattern. This means that the pattern is stored in sequential memory locations stored as (**maxy** - **yl** + 1) rows of **bytecnt** bytes per row. These patterns must obey the same rules as **BitMaps**. This means that they must consist of an even number of bytes per row and must be stored in memory beginning at a legal *word* address. (The mask for **BltPattern()** does not have to be in Chip RAM, though.)

Extracting from a Bit-packed Array

You use the routine **BltTemplate()** to extract a rectangular area from a source area and place it into a destination area. The following figure shows an example.

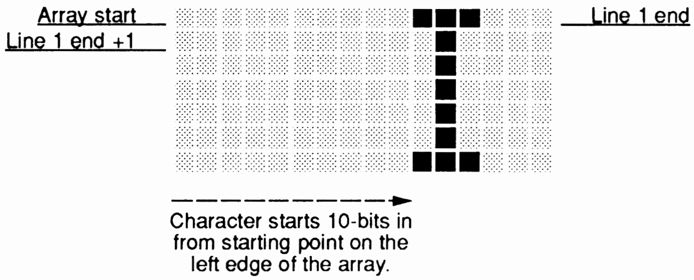


Figure 27-18: Example of Extracting from a Bit-Packed Array

For a rectangular bit array to be extracted from within a larger, rectangular bit array, the system must know how the larger array is organized. For this extraction to occur properly, you also need to tell the system the modulo for the inner rectangle. The modulo is the value that must be added to the address pointer so that it points to the correct word in the next line in this rectangularly organized array.

The following figure represents a single bitplane and the smaller rectangle to be extracted. The modulo in this instance is 4, because at the end of each line, you must add 4 to the address pointer to make it point to the first word in the smaller rectangle.

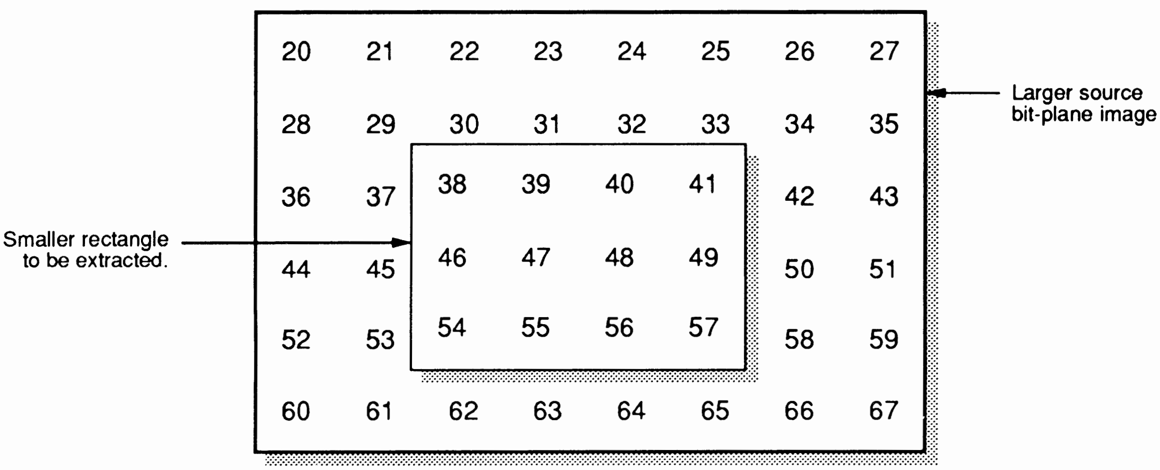


Figure 27-19: Modulo

Warning: The modulo value must be an even number of bytes.

BltTemplate() takes the following arguments:

```
BltTemplate(source, srcX, srcMod, &destRastPort, destX, destY, sizeX, sizeY);
```

The **source** argument specifies the rectangular bit array to use as the source template. Set this to the address of the nearest word (rounded down) that contains the first line of the source rectangle. The **srcX** argument gives the exact bit position (0-15) within the source address at which the rectangular bit array begins. The **srcMod** argument sets the source modulo so the next line of the rectangular bit array can be found.

The data from the source rectangle is copied into the destination **RastPort** specified by **destRastPort**. The **destX** and **destY** arguments indicate where the data from the source rectangle should be positioned within the destination **RastPort**. The **sizeX** and **sizeY** arguments indicate the dimensions of the data to be moved.

BltTemplate() uses **FgPen**, **BgPen**, **DrawMode** and **Mask** to place the template into the destination area. This routine differs from **BltPattern()** in that only a solid color is deposited in the destination drawing area, with or without a second solid color as the background (as in the case of text). Also, the template can be arbitrarily bit-aligned and sized in x.

Copying Rectangular Areas

Four routines use the blitter to copy rectangular areas from one section of a **BitMap** to another: **BltBitMap()**, **BltBitMapRastPort()**, **BltMaskBitMapRastPort()**, and **ClipBlit()**. All four of these blitter routines take a special argument called a **minterm**.

The **minterm** variable is an unsigned byte value which represents an action to be performed during the move. Since all the blitter routines uses the hardware blitter to move the data, they can take advantage of the blitter's ability to logically combine or change the data as the move is made. The most common operation is a direct copy from source area to destination, which uses a **minterm** set to hex value C0.

You can determine how to set the **minterm** variable by using the logic equations shown in the following tables. B represents data from the source rectangle and C represents data in the destination area.

Table 27-7: Minterm Logic Equations

Leftmost 4 Bits of MinTerm	Logic Term Included in Final Output
8	$B\underline{C}$
4	$\underline{B}C$
2	$\underline{B}\underline{C}$
1	BC

You can combine values to select the logic terms. For instance a **minterm** value of 0xC0 selects the first two logic terms in the table above. These logic terms specify that in the final destination area you will have data that occurs in source B only. Thus, C0 means a direct copy. The logic equation for this is:

$$BC + B\bar{C} = B(C + \bar{C}) = B$$

Logic equations may be used to decide on a number of different ways of moving the data. For your convenience, a few of the most common ones are listed below.

Table 27-8: Some Common MinTerm Values to Use for Copying

MinTerm Value	Logic Operation Performed During Copy
30	Replace destination area with inverted source B.
50	Replace destination area with an inverted version of itself.
60	Put B where C is not, put C where B is not (cookie cut).
80	Only put bits into destination where there is a bit in the same position for both source and destination (sieve operation).
C0	Plain vanilla copy from source B to destination C.

The graphics library blitter routines all accept a **minterm** argument as described above. **BltBitMap()** is the basic blitter routine, moving data from one **BitMap** to another.

BltBitMap() allows you to define a rectangle within a source **BitMap** and copy it to a destination area of the same size in another (or even the same) **BitMap**. This routine is used by the graphics library itself for rendering. **BltBitMap()** returns the number of planes actually involved in the blit. The syntax for the function is:

```
ULONG planes;

planes = BltBitMap(&srcBM, srcX, srcY, &dstBM, dstX, dstY,
            sizeX, sizeY, minterm, mask, tempA);
```

The source bitmap is specified by the **&srcBM** argument. The position of the source area within the bitmap is specified by **srcX** and **srcY**. The destination bitmap is specified by the **&dstBM** argument. The position of the destination area within the bitmap is specified by **dstX** and **dstY**.

The dimensions (in pixels) of the area to be moved is indicated by the **sizeX** and **sizeY** arguments. With the original custom chip set, the blitter size limits are 992 x 1024. With ECS the blitter size limits are 32,736 x 32,768. See the section on ''Determining Chip Versions'' earlier in this chapter to find out how to tell if the host system has ECS installed.

The **minterm** argument determines what logical operation to perform on the rectangle data as bits are moved (described above). The **mask** argument, normally set to 0xff, specifies which bitplanes will be involved in the blit operation and which will be ignored. If a bit is set in the **mask** byte, the corresponding bitplane is included. The **tempA** argument applies only to blits that overlap and, if non-NULL, points to Chip memory the system will use for temporary storage during the blit.

BltBitMapRastPort() takes most of the same arguments as **BltBitMap()**, but its destination is a **RastPort** instead of a **BitMap**. The syntax for the function is:

```
VOID BltBitMapRastPort(&srcBM, srcX, srcY, &dstRP, dstX, dstY,
        sizeX, sizeY, minterm);
```

The arguments here are the same as for **BltBitMap()** above. Note that the **BltBitMapRastPort()** function will respect the **RastPort.Mask** field. Only the planes specified in the **Mask** will be included in the operation.

A third type of blitter operation is provided by the **BltMaskBitMapRastPort()** function. This works the same as **BltBitMapRastPort()** except that it takes one extra argument, a pointer to a single bitplane mask of the same height and width as the source. The mask acts as a filter for the operation—a blit only occurs where the mask plane is non-zero. The syntax for the function is:

```
VOID BltMaskBitMapRastPort(&srcBM, srcX, srcY, &dstRP, dstX, dstY,
        sizeX, sizeY, minterm, bltmask);
```

The **bltmask** argument points to a word-aligned mask bitplane in Chip memory with the same dimensions as the source bitmap. Note that this function ignores the **Mask** field of the destination **RastPort**.

ClipBlit() takes most of the same arguments as the other blitter calls described above but it works with source and destination **RastPorts** and their layers. Before **ClipBlit()** moves data, it looks at the area from which and to which the data is being copied (**RastPorts**, not **BitMaps**) and determines if there are overlapping areas involved. If so, it splits up the overall operation into a number of bitmaps to move the data in the way you request. To call **ClipBlit()** use:

```
VOID ClipBlit(&srcRP, srcX, srcY, &dstRP, dstX, dstY, XSize, YSize, minterm);
```

Since **ClipBlit()** respects the **Layer** of the source and destination **RastPort**, it is the easiest blitter movement call to use with Intuition windows. The following code fragments show how to save and restore an undo buffer using **ClipBlit()**.

```
/* Save work rastport to an undo rastport */
ClipBlit(&drawRP, 0, 0, &undoRP, 0, 0, areaWidth, areaHeight, 0xC0);

/* restore undo rastport to work rastport */
ClipBlit(&undoRP, 0, 0, &drawRP, 0, 0, areaWidth, areaHeight, 0xC0);
```

Scaling Rectangular Areas

BitMapScale() will scale a single bitmap any integral size up to 16,383 times its original size. This function is available only in Release 2 and later versions of the OS. It is called with the address of a **BitScaleArgs** structure (see <*graphics/scale.h*>).

```
void BitMapScale(struct BitScaleArgs *bsa)
```

The **bsa** argument specifies the **BitMaps** to use, the source and destination rectangles, as well as the scaling factor. The source and destination may *not* overlap. The caller must ensure that the destination **BitMap** is large enough to receive the scaled-up copy of the source rectangle. The function **ScalerDiv()** is provided to help in the calculation of the destination **BitMap**'s size.

When to Wait for the Blitter

This section explains why you might have to call **WaitBlit()**, a special graphics function that suspends your task until the blitter is idle. Many of the calls in the graphics library use the Amiga's hardware blitter to perform their operation, most notably those which render text and images, fill or pattern, draw lines or dots and move blocks of graphic memory.

Internally, these graphics library functions operate in a loop, doing graphic operations with the blitter one plane at a time as follows:

```
OwnBlitter();          /* Gain exclusive access to the hardware blitter       */

for(planes=0; planes < bitmap->depth; planes++)
      {
      WaitBlit();     /* Sleep until the previous blitter operation completes */
      /* start a blit */
      }

DisownBlitter();       /* Release exclusive access to the hardware blitter    */
```

Graphics library functions that are implemented this way always wait for the blitter at the start and exit right after the final blit is *started*. It is important to note that when these blitter-using functions return to your task, the final (or perhaps only) blit has just been *started*, but not necessarily *completed*. This is efficient if you are making many such calls in a row because the next graphics blitter call always waits for the previous blitter operation to complete before starting its first blit.

However, if you are intermixing such graphics blitter calls with other code that accesses the same graphics memory then you must first **WaitBlit()** to make sure that the final blit of a previous graphics call is complete before you use any of the memory. For instance, if you plan to immediately deallocate or reuse any of the memory areas which were passed to your most recent blitter-using function call as a source, destination, or mask, it is imperative that you first call **WaitBlit()**.

> *Warning:* If you do not follow the above procedure, you could end up with a program that works correctly most of the time but crashes sometimes. Or you may run into problems when your program is run on faster machines or under other circumstances where the blitter is not as fast as the processor.

Accessing the Blitter Directly

To use the blitter directly, you must first be familiar with how its registers control its operation. This topic is covered thoroughly in the *Amiga Hardware Reference Manual* and is not repeated here. There are two basic approaches you can take to perform direct programming of the blitter: synchronous and asynchronous.

- Synchronous programming of the blitter is used when you want to do a job with the blitter right away. For synchronous programming, you first get exclusive access to the blitter with **OwnBlitter()**. Next call **WaitBlit()** to ensure that any previous blitter operation that might have been in progress is completed. Then set up your blitter operation by programming the blitter registers. Finally, start the blit and call **DisownBlitter()**.

- Asynchronous programming of the blitter is used when the blitter operation you want to perform does not have to happen immediately. In that case, you can use the **QBlit()** and **QBSBlit()** functions in order to queue up requests for the use of the blitter on a non-exclusive basis. You share the blitter with system tasks.

Whichever approach you take, there is one rule you should generally keep in mind about using the blitter directly:

> *Don't Tie Up The Blitter.* The system uses the blitter extensively for disk and display operation. While your task is using the blitter, many other system processes will be locked out. Therefore, use it only for brief periods and relinquish it as quickly as possible.

To use **QBlit()** and **QBSBlit()**, you must create a data structure called a **bltnode** (blitter node) that contains a pointer to the blitter code you want to execute. The system uses this structure to link blitter usage requests into a first-in, first-out (FIFO) queue. When your turn comes, your own blitter routine can be repeatedly called until your routine says it is finished using the blitter.

Two separate blitter queues are maintained. One queue is for the **QBlit()** routine. You use **QBlit()** when you simply want something done and you do not necessarily care when it happens. This may be the case when you are moving data in a memory area that is not currently being displayed.

The second queue is maintained for **QBSBlit()**. QBS stands for "queue-beam-synchronized". **QBSBlit()** requests form a beam-synchronized FIFO queue. When the video beam gets to a predetermined position, your blitter routine is called. Beam synchronization takes precedence over the simple FIFO. This means that if the beam sync matches, the beam-synchronous blit will be done before the non-synchronous blit in the first position in the queue. You might use **QBSBlit()** to draw into an area of memory that is currently being displayed to modify memory that has already been "passed-over" by the video beam. This avoids display flicker as an area is being updated.

The sole input to both **QBlit()** and **QBSBlit()** is a pointer to a **bltnode** data structure, defined in the include file *<hardware/blit.h>*. Here is a copy of the structure, followed by details about the items you must initialize:

```
struct bltnode
{
    struct  bltnode *n;
    int     (*function)();
    char    stat;
    short   blitsize;
    short   beamsync;
    int     (*cleanup)();
};
```

struct bltnode *n;

This is a pointer to the next **bltnode**, which, for most applications will be zero. You should not link **bltnodes** together. This is to be performed by the system in a separate call to **QBlit()** or **QBSBlit()**.

int (*function)( );

This is the address of your blitter function that the blitter queuer will call when your turn comes up. Your function must be formed as a subroutine, with an RTS instruction at the end. Follow Amiga programming conventions by placing the return value in D0 (or in C, use **return(value)**).

If you return a nonzero value, the system will call your routine again next time the blitter is idle until you finally return 0. This is done so that you can maintain control over the blitter; for example, it allows you to handle all five bitplanes if you are blitting an object with 32 colors. For display purposes, if you are blitting multiple objects and then saving and restoring the background, you must be sure that all planes of the object are positioned before another object is overlaid. This is the reason for the lockup in the blitter queue; it allows all work per object to be completed before going on to the next one.

Note: Not all C compilers can handle ***function()** properly! The system actually tests the processor *status codes* for a condition of equal-to-zero (Z flag set) or not-equal-to-zero (Z flag clear) when your blitter routine returns. Some C compilers do not set the processor status code properly (i.e., according to the value returned), thus it is not possible to use such compilers to write the (***function**)()) routine. In that case assembly language should be used. Blitter functions are normally written in assembly language anyway so they can take advantage of the ability of **QBlit()** and **QBSBlit()** to pass them parameters in processor registers.

The register passing conventions for these routines are as follows. Register A0 receives a pointer to the system hardware registers so that all hardware registers can be referenced as an offset from that address. Register A1 contains a pointer to the current **bltnode**. You may have queued up multiple blits, each of which perhaps uses the same blitter routine. You can access the data for this particular operation as an offset from the value in A1. For instance, a typical user of these routines can precalculate the blitter register values to be placed in the blitter registers and, when the routine is called, simply copy them in. For example, you can create a new structure such as the following:

```
INCLUDE "exec/types.i"
INCLUDE "hardware/blit.i"

STRUCTURE mybltnode,0
                ; Make this new structure compatible with a bltnode
                ; by making the first element a bltnode structure.
STRUCT bltnode,bn_SIZEOF
        UWORD   bltcon1         ; Blitter control register 1.
        UWORD   fwmask          ; First and last word masks.
        UWORD   lwmask
        UWORD   bltmda          ; Modulos for sources a, b,and c.
        UWORD   bltmdb
        UWORD   bltmdc
        UWORD   any_more_data   ; add anything else you want
LABEL mbn_SIZEOF
```

Other forms of data structures are certainly possible, but this should give you the general idea.

char stat;

Tells the system whether or not to execute the clean-up routine at the end. This byte should be set to CLEANUP (0x40) if cleanup is to be performed. If not, then the **bltnode cleanup** variable can be zero.

short beamsync;

The value that should be in the VBEAM counter for use during a beam-synchronous blit before the **function()** is called. The system cooperates with you in planning when to start a blit in the routine **QBSBlit()** by not calling your routine until, for example, the video beam has already passed by the area on the screen into which you are writing. This is especially useful during single buffering of your displays. There may be time enough to write the object between scans of the video display. You will not be visibly writing while the beam is trying to scan the object. This avoids flicker (part of an old view of an object along with part of a new view of the object).

int (*cleanup)();

The address of a routine that is to be called after your last return from the **QBlit()** routine. When you finally return a zero, the queuer will call this subroutine (ends in RTS or **return()**) as the clean-up. Your first entry to the function may have dynamically allocated some memory or may have done something that must be undone to make for a clean exit. This routine must be specified.

User Copper Lists

The Copper coprocessor allows you to produce mid-screen changes in certain hardware registers in addition to changes that the system software already provides. For example, it is the Copper that allows the Amiga to split the viewing area into multiple draggable screens, each with its own independent set of colors.

To create your own mid-screen effects on the system hardware registers, you provide "user Copper lists" that can be merged into the system Copper lists.

In the **ViewPort** data structure there is a pointer named **UCopIns**. If this pointer value is non-NULL, it points to a user Copper list that you have dynamically allocated and initialized to contain your own special hardware-stuffing instructions.

You allocate a user Copper list by an instruction sequence such as the following:

```
struct UCopList *uCopList = NULL;

/* Allocate memory for the Copper list.  Make certain that the initial */
/* memory is cleared.                                                  */
uCopList = (struct UCopList *)
        AllocMem(sizeof(struct UCopList), MEMF_PUBLIC|MEMF_CLEAR);

if (uCopList == NULL)
        return(FALSE);
```

> *Note:* User Copper lists do *not* have to be in Chip RAM.

COPPER LIST MACROS

Once this pointer to a user Copper list is available, you can use it with system macros (*<graphics/gfxmacros.h>*) to instruct the system what to add to its own list of things for the Copper to do within a specific **ViewPort**. The file *<graphics/gfxmacros.h>* provides the following five macro functions that implement user Copper instructions.

CINIT initializes the Copper list buffer. It is used to specify how many instructions are going to be placed in the Copper list. It is called as follows.

```
CINIT(uCopList, num_entries);
```

The **uCopList** argument is a pointer tot he user Copper list and **num_entries** is the number of entries in the list.

CWAIT waits for the video beam to reach a particular horizontal and vertical position. Its format is:

```
CWAIT(uCopList, v, h)
```

Again, **uCopList** is the pointer to the Copper list. The **v** argument is the vertical position for which to wait, specified relative to the top of the **ViewPort**. The legal range of values (for both NTSC and PAL) is from 0 to 255; **h** is the horizontal position for which to wait. The legal range of values (for both NTSC and PAL) is from 0 to 226.

CMOVE installs a particular value into a specified system register. Its format is:

```
CMOVE(uCopList, reg, value)
```

Again, **uCopList** is the pointer to the Copper list. The **reg** argument is the register to be affected, specified in this form: **custom.register-name** where the register-name is one of the registers listed in the **Custom** structure in <hardware/custom.h>. The **value** argument to **CMOVE** is the value to place in the register.

CBump increments the user Copper list pointer to the next position in the list. It is usually invoked for the programmer as part of the macro definitions **CWAIT** or **CMOVE**. Its format is:

```
CBump(uCopList)
```

where **uCopList** is the pointer to the user Copper list.

CEND terminates the user Copper list. Its format is:

```
CEND(uCopList)
```

where **uCopList** is the pointer to the user Copper list.

Executing any of the user Copper list macros causes the system to dynamically allocate special data structures called intermediate Copper lists that are linked into your user Copper list (the list to which **uCopList** points) describing the operation. When you call the function **MrgCop(&view)** as shown in the section called "Forming A Basic Display," the system uses all of its intermediate Copper lists to sort and merge together the real Copper lists for the system (**LOFCprList** and **SHFCprList**).

When your program exits, you must return to the system all of the memory that you allocated or caused to be allocated. This means that you must return the intermediate Copper lists, as well as the user Copper list data structure. Here are two different methods for returning this memory to the system.

```
/* Returning memory to the system if you have NOT
 * obtained the ViewPort from Intuition.  */
FreeVPortCopLists(viewPort);

/* Returning memory to the system if you HAVE
 * obtained the ViewPort from Intuition.  */
CloseScreen(screen);    /* Intuition only */
```

User Copper lists may be clipped, under Release 2 and later, to **ViewPort** boundaries if the appropriate tag (VTAG_USERCLIP_SET) is passed to **VideoControl()**. Under earlier releases, the user Copper list would "leak" through to lower **ViewPorts**.

COPPER LIST EXAMPLE

The example program below shows the use of user Copper lists under Intuition.

```
/*  UserCopperExample.c
    User Copper List Example
    For SAS/C 5.10a,
    compile with:  LC -b1 -cfist -L -v -y UserCopperExample.c
    link with lc.lib and amiga.lib
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/gfxbase.h>
```

```
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <graphics/videocontrol.h>
#include <intuition/intuition.h>
#include <intuition/preferences.h>
#include <hardware/custom.h>
#include <libraries/dos.h>

#include <clib/exec_protos.h>       /*  Prototypes.  */
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
#include <clib/dos_protos.h>

#include <stdlib.h>

/*  Use this structure to gain access to the custom registers.  */
extern struct Custom far custom;

/*  Global variables.  */
struct GfxBase        *GfxBase = NULL;
struct IntuitionBase  *IntuitionBase = NULL;
struct Screen         *screen = NULL;
struct Window         *window = NULL;

VOID main( VOID ), cleanExit( WORD );
WORD openAll( VOID ), loadCopper( VOID );


/*
 *   The main() routine -- just calls subroutines
 */
VOID main( VOID )
{
WORD ret_val;
struct IntuiMessage     *intuiMessage;

      /*  Open the libraries, a screen and a window.  */
      ret_val = openAll();
      if (RETURN_OK == ret_val)
      {
            /*  Create and attach the user Copper list.  */
            ret_val = loadCopper();
            if (RETURN_OK == ret_val)
            {
                  /*  Wait until the user clicks in the close gadget.  */
                  (VOID) Wait(1<<window->UserPort->mp_SigBit);

                  while (intuiMessage = (struct IntuiMessage *)GetMsg(window->UserPort))
                        ReplyMsg((struct Message *)intuiMessage);
            }
      }
      cleanExit(ret_val);
}


/*
 * openAll() -- opens the libraries, screen and window
 */
WORD openAll( VOID )
{
#define MY_WA_WIDTH 270 /*  Width of window.  */

      WORD ret_val = RETURN_OK;

      /*  Prepare to explicitly request Topaz 60 as the screen font.  */
      struct TextAttr topaz60 =
      {
            (STRPTR)"topaz.font",
            (UWORD)TOPAZ_SIXTY, (UBYTE)0, (UBYTE)0
      };

      GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 37L);
      if (GfxBase == NULL)
            ret_val = ERROR_INVALID_RESIDENT_LIBRARY;
      else
      {
```

```
        IntuitionBase = (struct IntuitionBase *)
              OpenLibrary("intuition.library", 37L);

        if (IntuitionBase == NULL)
              ret_val = ERROR_INVALID_RESIDENT_LIBRARY;
        else
        {
              screen = OpenScreenTags( NULL,
                       SA_Overscan, OSCAN_STANDARD,
                       SA_Title,    "User Copper List Example",
                       SA_Font,     (ULONG)&topaz60,
                       TAG_DONE);

              if (NULL == screen)
                    ret_val = ERROR_NO_FREE_STORE;
              else
              {
                    window = OpenWindowTags( NULL,
                             WA_CustomScreen, screen,
                             WA_Title,        "<- Click here to quit.",
                             WA_IDCMP,        CLOSEWINDOW,
                             WA_Flags,        WINDOWDRAG|WINDOWCLOSE|INACTIVEWINDOW,
                             WA_Left,         (screen->Width-MY_WA_WIDTH)/2,
                             WA_Top,          screen->Height/2,
                             WA_Height,       screen->Font->ta_YSize + 3,
                             WA_Width,        MY_WA_WIDTH,
                             TAG_DONE);

                    if (NULL == window)
                          ret_val = ERROR_NO_FREE_STORE;
              }
        }
  }

  return(ret_val);
}


/*
 * loadCopper() -- creates a Copper list program and adds it to the system
 */
WORD loadCopper( VOID )
{
register USHORT   i, scanlines_per_color;
         WORD     ret_val    = RETURN_OK;
struct   ViewPort *viewPort;
struct   UCopList *uCopList  = NULL;
struct   TagItem  uCopTags[] =
          {
                { VTAG_USERCLIP_SET, NULL },
                { VTAG_END_CM, NULL }
          };

UWORD    spectrum[] =
          {
                0x0604, 0x0605, 0x0606, 0x0607, 0x0617, 0x0618, 0x0619,
                0x0629, 0x072a, 0x073b, 0x074b, 0x074c, 0x075d, 0x076e,
                0x077e, 0x088f, 0x07af, 0x06cf, 0x05ff, 0x04fb, 0x04f7,
                0x03f3, 0x07f2, 0x0bf1, 0x0ff0, 0x0fc0, 0x0ea0, 0x0e80,
                0x0e60, 0x0d40, 0x0d20, 0x0d00
          };

#define NUMCOLORS 32

      /*  Allocate memory for the Copper list.  */
      /*  Make certain that the initial memory is cleared.  */
      uCopList = (struct UCopList *)
            AllocMem(sizeof(struct UCopList), MEMF_PUBLIC|MEMF_CLEAR);

      if (NULL == uCopList)
            ret_val = ERROR_NO_FREE_STORE;
      else
      {
            /*  Initialize the Copper list buffer.  */
            CINIT(uCopList, NUMCOLORS);
```

```
            scanlines_per_color = screen->Height/NUMCOLORS;

            /*  Load in each color.  */
            for (i=0; i<NUMCOLORS; i++)
                  {
                  CWAIT(uCopList, (i*scanlines_per_color), 0);
                  CMOVE(uCopList, custom.color[0], spectrum[i]);
                  }

            CEND(uCopList);    /*  End the Copper list  */

            viewPort = ViewPortAddress(window); /*  Get a pointer to the ViewPort.  */
            Forbid();    /*  Forbid task switching while changing the Copper list.  */
            viewPort->UCopIns=uCopList;
            Permit();    /*  Permit task switching again.  */

            /*  Enable user copper list clipping for this ViewPort.  */
            (VOID) VideoControl( viewPort->ColorMap, uCopTags );

            RethinkDisplay(); /*  Display the new Copper list.  */

            return(ret_val);
      }
}


/*
 *  cleanExit() -- returns all resources that were used.
 */
VOID cleanExit( WORD retval )
{
struct ViewPort *viewPort;

if (NULL != IntuitionBase)
{
      if (NULL != screen)
      {
            if (NULL != window)
            {
                  viewPort = ViewPortAddress(window);
                  if (NULL != viewPort->UCopIns)
                  {
                        /*  Free the memory allocated for the Copper.  */
                        FreeVPortCopLists(viewPort);
                        RemakeDisplay();
                  }
                  CloseWindow(window);
            }
            CloseScreen(screen);
      }
      CloseLibrary((struct Library *)IntuitionBase);
}

if (NULL != GfxBase)
      CloseLibrary((struct Library *)GfxBase);

exit((int)retval);
}
```

ECS and Genlocking Features

The Enhanced Chip Set (ECS) Denise chip (8373-R2a), coupled with the Release 2 graphics library, opens up a whole new set of genlocking possibilities. Unlike the old Denise, whose only genlocking ability allowed keying on color register zero, the ECS Denise allows keying on any color register. Also, the ECS Denise allows keying on any bitplane of the **ViewPort** being genlocked. With the ECS Denise, the border area surrounding the display can be made transparent (always passes video) or opaque (overlays using color 0). All the new features are set individually for each **ViewPort**. These features can be used in conjunction with each other, making interesting scenarios possible.

GENLOCK CONTROL

Using **VideoControl()**, a program can enable, disable, or obtain the state of a **ViewPort**'s genlocking features. It returns NULL if no error occurred. The function uses a tag based interface:

```
error = BOOL VideoControl( struct ColorMap *cm, struct TagItem *ti );
```

The **ti** argument is a list of video commands stored in an array of **TagItem** structures. The **cm** argument specifies which **ColorMap** and, indirectly, which **ViewPort** these genlock commands will be applied to. The possible commands are:

```
VTAG_BITPLANEKEY_GET, _SET, _CLR
VTAG_CHROMA_PLANE_GET, _SET
VTAG_BORDERBLANK_GET, _SET, _CLR
VTAG_BORDERNOTRANS_GET, _SET, _CLR
VTAG_CHROMAKEY_GET, _SET, _CLR
VTAG_CHROMAPEN_GET, _SET, _CLR
```

This section covers only the genlock **VideoControl()** tags. See *<graphics/videocontrol.h>* for a complete list of all the available tags you can use with **VideoControl()**.

VTAG_BITPLANEKEY_GET is used to find out the status of the bitplane keying mode. VTAG_BITPLANEKEY_SET and VTAG_BITPLANEKEY_CLR activate and deactivate bitplane keying mode. If bitplane key mode is on, genlocking will key on the bits set in a specific bitplane from the **ViewPort** (the specific bitplane is set with a different tag). The data portion of these tags is NULL.

For inquiry commands like VTAG_BITPLANEKEY_GET (tags ending in _GET), **VideoControl()** changes the _GET tag ID (**ti_Tag**) to the corresponding _SET or _CLR tag ID, reflecting the current state of the genlock mode. For example, when passed the following tag array:

```
struct TagItem videocommands[] =
{
    {VTAG_BITPLANEKEY_GET, NULL},
    {VTAG_END_CM, NULL}
};
```

VideoControl() changes the VTAG_BITPLANEKEY_GET tag ID (**ti_Tag**) to VTAG_BITPLANEKEY_SET if bitplane keying is currently on, or to VTAG_BITPLANEKEY_CLR if bitplane keying is off. In both of these cases, **VideoControl()** only uses the tag's ID, ignoring the tag's data field (**ti_Data**).

The VTAG_CHROMA_PLANE_GET tag returns the number of the bitplane keyed on when bitplane keying mode is on. **VideoControl()** changes the tag's data value to the bitplane number. VTAG_CHROMA_PLANE_SET sets the bitplane number to the tag's data value.

VTAG_BORDERBLANK_GET is used to obtain the border blank mode status. This tag works exactly like VTAG_BITPLANEKEY_GET. **VideoControl()** changes the tag's ID to reflect the current border blanking state. VTAG_BORDERBLANK_SET and VTAG_BORDERBLANK_CLR activate and deactivate border blanking. If border blanking is on, the Amiga will not display anything in its display border, allowing an external video signal to show through the border area. On the Amiga display, the border appears black. The data portion of these tags is NULL.

The VTAG_BORDERNOTRANS_GET, _SET and _CLR tags are used, respectively, to obtain the status of border-not-transparent mode, and to activate and to deactivate this mode. If set, the Amiga display's border will overlay external video with the color in register 0. Because border blanking mode takes precedence over border-not-transparent mode, setting border-not-transparent has no effect if border blanking is on. The data portion of these tags is NULL.

The VTAG_CHROMAKEY_GET, _SET and _CLR tags are used, respectively, to obtain the status of chroma keying mode, and to activate and deactivate chroma keying mode. If set, the genlock will key on colors from specific color registers (the specific color registers are set using a different tag). If chroma keying is not set, the genlock will key on color register 0. The data portion of these tags is NULL.

VTAG_CHROMAPEN_GET obtains the chroma keying status of an individual color register. The tag's **ti_Data** field contains the register number. Like the other _GET tags, **VideoControl()** changes the tag ID (**ti_Tag**) to one that reflects the current state of the mode. VTAG_CHROMAPEN_SET and VTAG_CHROMAPEN_CLR activate and deactivate chroma keying for each individual color register. Chroma keying can be active for more than one register. By turning off border blanking and activating chroma keying mode, but turning off chroma keying for each color register, a program can overlay every part of an external video source, completely blocking it out.

After using **VideoControl()** to set values in the **ColorMap**, the corresponding **ViewPort** has to be rebuilt with **MakeVPort()**, **MrgCop()** and **LoadView()**, so the changes can take effect. A program that uses a screen's **ViewPort** rather than its own **ViewPort** should use the Intuition functions **MakeScreen()** and **RethinkDisplay()** to make the display changes take effect.

The following code fragment shows how to access the genlock modes.

```
struct Screen *genscreen;
struct ViewPort *vp;
struct TagItem vtags [24];

      /* The complete example opened a window, rendered some colorbars,  */
      /* and added gadgets to allow the user to turn the various genlock */
      /* modes on and off.                                               */

      vp = &(genscreen->ViewPort);

      /* Ascertain the current state of the various modes. */

      /* Is borderblanking on? */
      vtags[0].ti_Tag = VTAG_BORDERBLANK_GET;
      vtags[0].ti_Data = NULL;

      /* Is bordertransparent set? */
      vtags[1].ti_Tag = VTAG_BORDERNOTRANS_GET;
      vtags[1].ti_Data = NULL;

      /* Key on bitplane? */
      vtags[2].ti_Tag = VTAG_BITPLANEKEY_GET;
      vtags[2].ti_Tag = NULL;

      /* Get plane which is used to key on */
      vtags[3].ti_Tag = VTAG_CHROMA_PLANE_GET;
      vtags[3].ti_Data = NULL;
```

```
    /* Chromakey overlay on? */
    vtags[4].ti_Tag = VTAG_CHROMAKEY_GET;
    vtags[4].ti_Data = NULL;

    for (i = 0; i < 16; i++)
    {
          /* Find out which colors overlay */
          vtags[i + 5].ti_Tag = VTAG_CHROMA_PEN_GET;
          vtags[i + 5].ti_Data = i;
    }

    /* Indicate end of tag array */
    vtags[21].ti_Tag = VTAG_END_CM;
    vtags[21].ti_Data = NULL;

    /* And send the commands. On return the Tags themselves will
    * indicate the genlock settings for this ViewPort's ColorMap.
    */
    error = VideoControl(vp->ColorMap, vtags);

    /* The complete program sets gadgets to reflect current states. */

    /* Will only send single commands from here on. */
    vtags[1].ti_Tag = VTAG_END_CM;

    /* At this point the complete program gets an input event and sets/clears the
       genlock modes as requested using the vtag list and VideoControl().
    */

    /* send video command */
    error = VideoControl(vp->ColorMap, vtags);

    /* Now use MakeScreen() and RethinkDisplay() to make the VideoControl()
    *  changes take effect.  If we were using our own ViewPort rather than
    * borrowing one from a screen, we would instead do:
    *
    *   MakeVPort(ViewAddress(),vp);
    *   MrgCop(ViewAddress());
    *   LoadView(ViewAddres());
    */
    MakeScreen(genscreen);
    RethinkDisplay();

/* The complete program closes and frees everything it had opened or allocated. */


/* The complete example calls the CheckPAL function, which is included below in its
   entirety for illustrative purposes.
*/

BOOL CheckPAL(STRPTR screenname)
{
    struct Screen *screen;
    ULONG modeID = LORES_KEY;
    struct DisplayInfo displayinfo;
    BOOL IsPAL;

    if (GfxBase->LibNode.lib_Version >= 36)
    {
          /*
          * We got at least V36, so lets use the new calls to find out what
          * kind of videomode the user (hopefully) prefers.
          */

          if (screen = LockPubScreen(screenname))
          {
                /*
                * Use graphics.library/GetVPModeID() to get the ModeID of the specified screen.
                * Will use the default public screen (Workbench most of the time) if NULL It is
                * _very_ unlikely that this would be invalid, heck it's impossible.
                */
                if ((modeID = GetVPModeID(&(screen->ViewPort))) != INVALID_ID)
                {
                      /*
                      * If the screen is in VGA mode, we can't tell whether the system is PAL
                      * or NTSC. So to be foolproof we fall back to the displayinfo of the default
```

```
                * monitor by inquiring about just the LORES_KEY displaymode if we don't know.
                * The default.monitor reflects the initial video setup of the system, thus
                * for either ntsc.monitor or pal.monitor.  We only use the displaymode of the
                * is an alias specified public screen if it's display mode is PAL or NTSC and
                * NOT the default.
                */
                if (!((modeID & MONITOR_ID_MASK) == NTSC_MONITOR_ID ||
                (modeID & MONITOR_ID_MASK) == PAL_MONITOR_ID))
                modeID = LORES_KEY;
            }
            UnlockPubScreen(NULL, screen);
        } /* if fails modeID = LORES_KEY. Can't lock screen, so fall back on default monitor. */

        if (GetDisplayInfoData(NULL, (UBYTE *) & displayinfo,
        sizeof(struct DisplayInfo), DTAG_DISP, modeID))
        {
            if (displayinfo.PropertyFlags & DIPF_IS_PAL)
                IsPAL = TRUE;
            else
                IsPAL = FALSE;
            /* Currently the default monitor is always either PAL or NTSC. */
        }
    }
    else
        /* < V36. The enhancements to the videosystem in V36 (and above) cannot be better
        * expressed than with the simple way to determine PAL in V34.
        */
        IsPAL= (GfxBase->DisplayFlags & PAL) ? TRUE : FALSE;

    return(IsPAL);
}
```

Function Reference

The following are brief descriptions of the Amiga's graphics primitives. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 27-9: Graphics Primitives Functions

Display Set-up Functions	Description
InitView()	Initializes the **View** structure.
InitBitMap()	Initializes the **BitMap** structure.
RASSIZE()	Calculates the size of a **ViewPort**'s **BitMap**.
AllocRaster()	Allocates the bitplanes needed for a **BitMap**.
FreeRaster()	Frees the bitplanes created with **AllocRaster()**.
InitVPort()	Initializes the **ViewPort** structure.
GetColorMap()	Returns the **ColorMap** structure used by **ViewPorts**.
FreeColorMap()	Frees the **ColorMap** created by **GetColorMap()**.
LoadRGB4()	Loads the color registers for a given **ViewPort**.
SetRGB4CM()	Loads an individual color register for a given **ViewPort**.
MakeVPort()	Creates the intermediate Copper list program for a **ViewPort**.
MrgCop()	Merges the intermediate Copper lists.
LoadView()	Displays a given **View**.
FreeCprList()	Frees the Copper list created with **MrgCop()**
FreeVPortCopLists()	Frees the intermediate Copper lists created with **MakeVPort()**.
OFF_DISPLAY()	Turns the video display DMA off
ON_DISPLAY()	Turns the video display DMA back on again.

Release 2 Display Set-up Functions	Description
FindDisplayInfo()	Returns the display database handle for a given ModeID (V36).
GetDisplayInfoData()	Looks up a display attribute in the display database (V36).
VideoControl()	Sets, clears and gets the attributes of an existing display (V36).
GfxNew()	Creates **ViewExtra** or **ViewPortExtra** used in Release 2 displays (V36).
GfxAssociate()	Attaches a **ViewExtra** to a **View** (V36).
GfxFree()	Frees the **ViewExtra** or **ViewPortExtra** created by **GfxNew()** (V36).
OpenMonitor()	Returns the **MonitorSpec** structure used in Release 2 **Views** (V36).
CloseMonitor()	Frees the **MonitorSpec** structure created by **OpenMonitor()** (V36).
GetVPModeID()	Returns the Release 2 ModeID of an existing **ViewPort** (V36).
ModeNotAvailable()	Determines if a display mode is available from a given ModeID (V36).

Drawing Functions	Description
InitRastPort()	Initialize a **RastPort** structure.
InitArea()	Initialize the **AreaInfo** structure used with a **RastPort**.
SetWrMask()	Set the **RastPort.Mask**.
SetAPen()	Set the **RastPort.FgPen** foreground pen color.
SetBPen()	Set the **RastPort.BgPen** background pen color.
SetOPen()	Set the **RastPort.AOlPen** area fill outline pen color.
SetDrMode()	Set the **RastPort.DrawMode** drawing mode.
SetDrPt()	Set the **RastPort.LinePtrn** line drawing pattern.
SetAfPt()	Set the **RastPort** area fill pattern and size.
WritePixel()	Draw a single pixel in the foreground color at a given coordinate.
ReadPixel()	Find the color of the pixel at a given coordinate.
DrawCircle()	Draw a circle with a given radius and center point.
DrawEllipse()	Draw an ellipse with the given radii and center point.
Move()	Move the **RastPort** drawing pen to a given coordinate.
Draw()	Draw a line from the current pen location to a given coordinate.
PolyDraw()	Draw a polygon with a given set of vertices.
AreaMove()	Set the anchor point for a filled polygon.
AreaDraw()	Add a new vertice to an area-fill polygon.
AreaEnd()	Close and area-fill polygon, draw it and fill it.
BNDRYOFF()	Turn off area-outline pen usage activated with **SetOPen()**.
AreaCircle()	Draw a filled circle with a given radius and center point.
AreaEllipse()	Draw a filled ellipse with the given radii and center point.
Flood()	Flood fill a region starting at a given coordinate.
RectFill()	Flood fill a rectangular area at a given location and size.

Data Movement Functions	Description
BltClear()	Use the hardware blitter to clear a block of memory.
SetRast()	Fill the **RastPort.BitMap** with a given color.
ScrollRaster()	Move a portion of a **RastPort.BitMap**.
BltPattern()	Draw a rectangular pattern of pixels into a **RastPort.BitMap**. The x-dimension of the rectangle must be word-aligned and word-sized.
BltTemplate()	Draw a rectangular pattern of pixels into a **RastPort.BitMap**. The x-dimension of the rectangle can be arbitrarily bit-aligned and sized.
BltBitMap()	Copy a rectangular area from one **BitMap** to a given coordinate in another **BitMap**.
BltBitMapRastPort()	Copy a rectangular area from a **BitMap** to a given coordinate in a **RastPort.BitMap**.
BltMaskBitMapRastPort()	Copy a rectangular area from a **BitMap** to a **RastPort.BitMap** through a mask bitplane.
ClipBlit()	Copy a rectangular area from one **RastPort** to another with respect to their **Layers**.
BitMapScale()	Scale a rectangular area within a **BitMap** to new dimensions (V36).

Hardware Programming Functions	Description
OwnBlitter()	Obtain exclusive access to the Amiga's hardware blitter.
DisownBlitter()	Relinquish exclusive access to the blitter.
WaitBlit()	Suspend until the current blitter operation has completed.
QBlit()	Place a **bltnode**-style asynchronous blitter request in the system queue
QBSBlit()	Place a **bltnode**-style asynchronous blitter request in the beam synchronized queue.
CINIT()	Initialize the user Copper list buffer.
CWAIT()	Instructs the Copper to wait for the video beam to reach a given position.
CMOVE()	Instructs the Copper to place a value into a given hardware register.
CBump()	Instructs the Copper to increment its Copper list pointer.
CEND()	Terminate the user Copper list.

Chapter 28
GRAPHICS SPRITES, BOBS AND ANIMATION

This chapter describes how to use the functions provided by the graphics library to manipulate and animate Graphic Elements (also called GELs). It is divided into six sections:

- An overview of the GELs animation system, including fundamental terms and structures
- Explanation of simple (hardware) Sprites and an example showing their usage
- Explanation of VSprites and an example showing their usage
- Explanation of Bobs and an example showing their usage
- Discussion of topics that apply to all GELs such as collision detection and data structure extensions.
- Discussion of animation, using **AnimComps** and **AnimObs** and an example showing their usage

About the GELs System

Before going into details, a quick glossary is in order. A *playfield* forms the background that GELs operate in. It encompasses the **View**, **ViewPort**, and **RastPort** data structures. (VSprites appear *over*, and Bobs appear *in* the playfield.) Playfields can be created and controlled at several levels. Refer to the ''Graphics Primitives'' and ''Layers Library'' chapters for details on lower-level playfield control. The ''Intuition Screens'' chapter explains how to get higher-level access to playfields.

GELs, or graphic elements, are special graphic objects that appear in the foreground and can be moved easily around the display. They are software constructs based on the Amiga's sprite and blitter hardware. The GELs system is compatible with all playfield modes, including dual-playfield. All the various types of GELs are defined by data structures found in *<graphics/gels.h>*.

TYPES OF GELS

The GEL types are (in order of increasing complexity):

VSprites	for Virtual Sprites. These are represented by the **VSprite** data structure and implemented with sprite hardware.
Bobs	Blitter Objects. These are represented by the **VSprite** and **Bob** data structures and implemented with blitter hardware.
AnimComps	Animation Components. These are represented by the **VSprite**, **Bob** and **AnimComp** data structures and implemented with blitter hardware.
AnimObs	Animation Objects. These are used to group AnimComps. They are not strictly GELs, but are described here.

Simple Sprites

Simple Sprites (also known as hardware sprites) are not really part of the GELs system but are the basis for VSprites. Simple Sprites are graphic objects implemented in hardware that are easy to define and easy to animate. The Amiga hardware has the ability to handle up to eight such sprite objects. Each Simple Sprite is produced by one of the Amiga's eight sprite DMA channels. They are 16-bits wide and arbitrarily tall.

The Amiga system software offers a choice of how to use these hardware sprites. After a sprite DMA channel has displayed the last line of a Simple Sprite, the system can reuse the channel for a different sprite lower on the screen. This is how VSprites are implemented—as a software construct based on the sprite hardware.

Hence, Simple Sprites are not really part of the animation system (they are *not* GELs). In fact, if Simple Sprites and GELs are used in the same display, the GELs system must be told specifically which Simple Sprites to avoid. Simple Sprites are described in this chapter because they are alternatives to VSprites.

VSprites

The VSprite, or virtual sprite, is the simplest type of GEL. The **VSprite** data structure contains just a bit more information than is needed to define a hardware sprite. VSprites take advantage of the system's ability to reuse sprite DMA channels—each VSprite can be temporarily assigned to a hardware sprite, as needed. This makes it appear to an application program that it has a *virtually* unlimited supply of VSprites.

Since VSprites are based on hardware sprites, rules that apply to hardware sprites apply to VSprites too. VSprites are not rendered into the underlying **BitMap** of the playfield and so do not affect any bits in the **BitMap**. Because they are hardware based, *they are positioned at absolute display coordinates and are not affected by the movement of screens*. The starting position of a sprite must not occur before scanline 20, because of certain hardware DMA time constraints. VSprites have the same size limitations as hardware sprites, they are 16-bits wide and arbitrarily tall.

The **VSprite** data structure also serves as the root structure of more complex GEL types—Bobs and AnimComps.

Bobs and AnimComps

Like VSprites, Bobs and AnimComps are graphics objects that make animation easier. They are rendered using the blitter. The blitter is a special Amiga hardware component used to move data quickly and efficiently, optionally performing logical operations as it does. It can be used to move any kind of data but is especially well suited to moving rectangular blocks of display data.

It is important to keep in mind that Bobs and AnimComps are based on the blitter hardware while VSprites use the sprite hardware. However all three GEL types use the **VSprite** structure as their root data structure. The system uses pointers to link the **VSprite**, **Bob** and **AnimComp** structures, "extending" the **VSprite** structure to include all GEL types.

Since Bobs and AnimComps are rendered with the blitter they actually change the underlying playfield **BitMap**. The **BitMap** area where the GEL is rendered can be saved. By moving the GEL to new locations in small increments while also saving and restoring the **Bitmap** as you proceed, you can create an animation effect. Bobs and AnimComps use the same coordinates as the playfield and can be any size.

AnimObs

The AnimOb (Animation Object) is a data structure that is used to group one or more AnimComps for convenient movement. For example, an AnimOb could be created that consists of two AnimComps, one that looks like a planet and another containing a sequence that describes orbiting moons. By moving just the AnimOb the image of the planet can be moved across the display and the moons will travel along with it, orbiting the planet the entire time. The system automatically manages the movement of all the AnimComps associated with the AnimOb.

VSprites vs. Bobs

If you are going to manage the movement and sequencing of GELS yourself, you need to decide if sprite animation (VSprites) or blitter animation (Bobs, AnimComps and AnimObs) best suit your needs. If you've got simple requirements or lots of coding time, you may even opt to use only Simple Sprites, and control them yourself. On the other hand if you want the system to manage your animations, AnimComps *must* be used and they are Bobs at heart.

Some fundamental differences between VSprites and Bobs are:

- VSprite images and coordinates currently low-resolution pixels, even on a high resolution display. Bob images and coordinates have the same resolution as the playfield they are rendered into.
- VSprites have a maximum width of 16 (low resolution) pixels. Bobs can be any width (although larg Bobs tend to slow down the system). The height of either VSprites or Bobs can be as tall as the display.
- VSprites have a maximum of three colors (Simple Sprites can have fifteen if they're attached). Because the system uses the Copper to control VSprite colors on the fly, the colors are not necessarily the same as those in the background playfield. Bobs can use any or all of the colors in the background playfield. Limiting factors include playfield resolution and display time. Bobs with more colors take longer to display.
- VSprites are positioned using absolute display coordinates, and don't move with screens. Bobs follow screen movement.

In general, VSprites offer speed, while Bobs offer flexibility.

The following figure shows how the various GEL data structures, **VSprites**, **Bobs** and **AnimComps** are linked together.

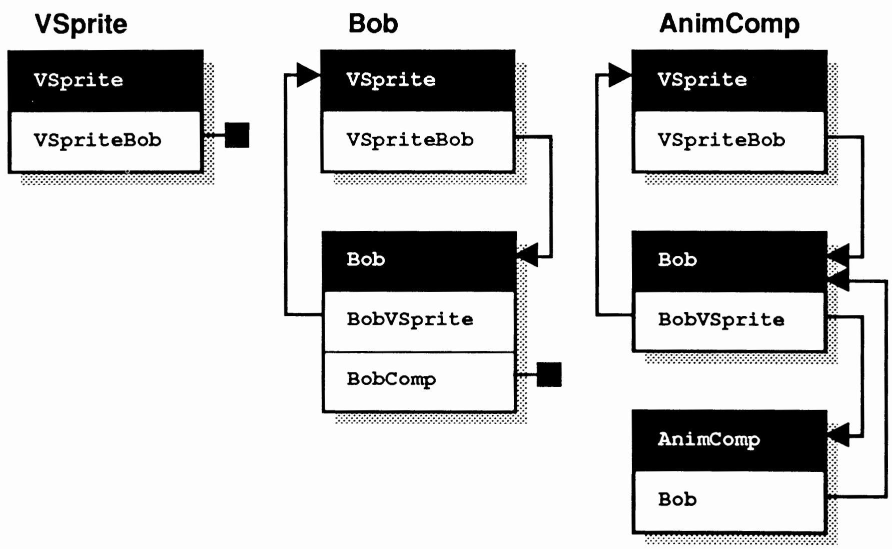


Figure 28-1: GEL Structure Layout

THE GELS SYSTEM

Before you can use the GELs system, you must set up a playfield. The GELs system requires access to a **View**, **ViewPort**, and **RastPort** structure. These structures may be set up through the graphics library or Intuition. For most examples in this chapter, the Intuition library is used for this purpose.

All GELs have a **VSprite** structure at their core. The system keeps track of all the GELs that it will display (the active GELs) by using a standard Exec list structure to link the VSprites. This list is accessed via the **GelsInfo** data structure, which in turn is associated with the **RastPort**. The **GelsInfo** structure is defined in the file *<graphics/rastport.h>*.

A new GEL is introduced to the system by calling **AddGel()** to link it into the **GelsInfo** list. The new GEL is added immediately ahead of the first existing GEL whose x, y value is greater than or equal to that of the new GEL, always trying to keep the list sorted.

As GELs are moved about the screen, their x, y values are constantly changing. **SortGList()** re-sorts this list by the x, y values. (Although this is a list of **VSprite** structures, bear in mind that some or all may really be Bobs or AnimComps.)

The basic set up of the **GelsInfo** structure requires three important fields: **sprRsrvd**, **gelHead** and **gelTail**. The **sprRsrvd** field tells the system which hardware sprites not to use when managing true VSprites. For instance, Intuition uses sprite 0 for the mouse pointer so this hardware sprite is not available for assignment to a VSprite. The **gelHead** and **gelTail** are **VSprite** structures that are used to manage the list of GELs. They are never displayed. To activate or deactivate a GEL, a system call is made to add it to or delete it from this list.

Other fields must be set up to provide for collision detection, color optimization, and other features. A complete example for setting up the **GelsInfo** structure is shown in the animtools.c lisitng at the end of this chapter.

Initializing the GEL System

To initialize the animation system, call the system function **InitGels()**. It takes the form:

```
struct VSprite  *vsHead;
struct VSprite  *vsTail;
struct GelsInfo *gInfo;

InitGels(vsHead, vsTail, gInfo);
```

The **vsHead** argument is a pointer to the **VSprite** structure to be used as the GEL list head. (You must allocate an actual **VSprite** structure for **vsHead** to point to.) The **vsTail** argument is a pointer to the **VSprite** structure to be used as the GEL list tail. (You must allocate an actual **VSprite** structure for **vsTail** to point to.) The **gInfo** argument is a pointer to the **GelsInfo** structure to be initialized.

InitGels() forms these structures into a linked list of GELs that is empty except for these two dummy elements (the head and tail). It gives the head **VSprite** the maximum negative x and y positions and the tail **VSprite** the maximum positive x and y positions. This is to aid the system in keeping the list sorted by x, y values, so GELs that are closer to the top and left of the display are nearer the head of the list. The memory space that the **VSprites** and **Gelsinfo** structures take up must already have been allocated. This can be done either by declaring them statically or explicitly allocating memory for them.

Once the **GelsInfo** structure has been allocated and initialized, GELs can be added to the system. Refer to the **setupGelSys()** and **cleanupGelsys()** functions in the animtools.c lisitng at the end of the chapter for examples of allocating, initializing and freeing a **GelsInfo** structure.

Using Simple (Hardware) Sprites

Simple Sprites can be used to create animations, so even though are not really part of the GELs system, they are described here as a possible alternative to using VSprites. For more information on the sprite hardware, including information on attached sprites, see the *Amiga Hardware Reference Manual*.

The **SimpleSprite** structure is found in *<graphics/sprite.h>*. It has fields that indicate the height and position of the Simple Sprite and a number that indicates which of the 8 hardware sprites to use.

Simple Sprites are always 16 bits wide, which is why there is no width member in the **SimpleSprite** structure. Currently, sprites *always* appear as low-resolution pixels, and their position is specified in the same way. If the sprite is being moved across a high-resolution display in single pixel increments, it will appear to move two high-resolution pixels for each increment. In low-resolution mode, a single Lores pixel movement will be seen. Similarly, in an interlaced display, the y direction motions are in two-line increments. The same sprite image data is placed into both even and odd fields of the interlaced display, so the sprite will appear to be the same size in any display mode.

The upper left corner of the **ViewPort** area has coordinates (0,0). Unlike VSprites, the motion of a Simple Sprite can be relative to this position. (That is, if you create a Simple Sprite relative to your **ViewPort** and move the **ViewPort** around, the Simple Sprite can move as well, but its movement is relative to the origin of the **ViewPort**.)

The Sprite pairs 0/1, 2/3, 4/5, and 6/7 share color registers. See "VSprite Advanced Topics" later in this chapter, for precautions to take if Simple Sprites and VSprites are used at the same time.

The following figure shows which color registers are used by Simple Sprites.

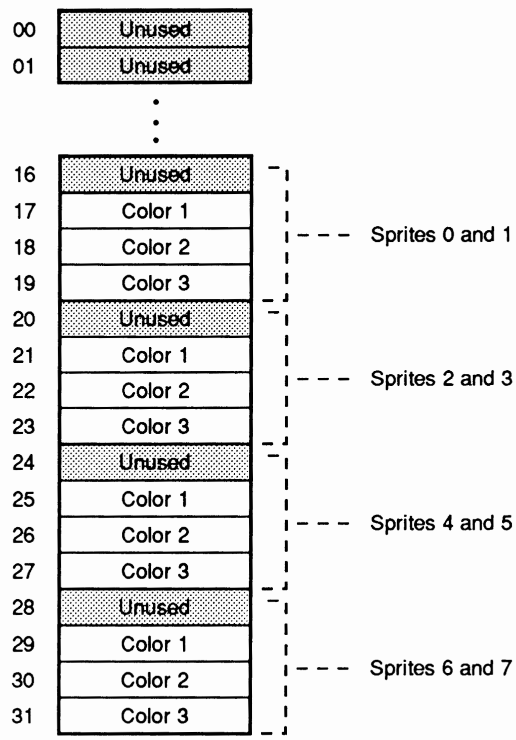


Figure 28-2: Sprite Color Registers

Sprites do not have *exclusive* use of the color registers. If the **ViewPort** is 5 bitplanes deep, all 32 of the system color registers will still be used by the playfield display hardware.

> *Note:* Color zero for all Sprites is always a "transparent" color, and the colors in registers 16, 20, 24, and 28 are not used by sprites. These colors will be seen only if they are rendered into a playfield. For further information, see the *Amiga Hardware Reference Manual.*

If there are two **ViewPorts** with different color sets on the same display, a sprite will switch colors when it is moved across their boundary. For example, Sprite 0 and 1 will appear in colors 17-19 of whatever **ViewPort** they happen to be over. This is because the system jams *all* the **ViewPort**'s colors into the display hardware at the top of each **ViewPort**.

SIMPLE SPRITE FUNCTIONS

There are four basic functions that you use to to control Simple Sprites:

GetSprite()	Attempts to allocates a sprite for exclusive use
ChangeSprite()	Modifies a Simple Sprite's image data
MoveSprite()	Changes a Simple Sprite's position
FreeSprite()	Relinquishes a sprite so it can be used by others

To use these Simple Sprite functions (or the VSprite functuons) the SPRITE flag must have been set in the **NewScreen** structure for **OpenScreen()**. If Intuition is not being used, this flag must be specified in the **View** and **ViewPort** data structures before **MakeVPort()** is called.

Accessing A Hardware Sprite

GetSprite() is used to gain exclusive access to one of the eight hardware sprites. Once you have gained control of a hardware sprite, it can no longer be allocated by the GELs system for use as a VSprite. The call is made like this:

```
struct SimpleSprite *sprite;
SHORT               number;

if (-1 == (sprite_num = GetSprite(sprite, number)))
    return_code = RETURN_WARN;  /* did not get the sprite */
```

The inputs to the **GetSprite()** function are a pointer to a **SimpleSprite** structure and the number (0-7) of the hardware sprite to be accessed, or -1 to get the first available sprite.

A value of 0-7 is returned if the request was granted, specifying which sprite was allocated. A returned value of -1 means the requested sprite was not available. If the call succeeds, the **SimpleSprite** data structure will have its sprite number field filled in with the appropriate number.

Changing The Appearance Of A Simple Sprite

The **ChangeSprite()** function can be used to alter the appearance of a Simple Sprite. **ChangeSprite()** substitutes new image data for the data currently used to display a Simple Sprite. It is called by the following sequence:

```
struct ViewPort     *vp;
struct SimpleSprite *sprite;
APTR                newdata;

ChangeSprite(vp, sprite, newdata);
```

The **vp** input to this function is a pointer to the **ViewPort** for this Sprite or 0 if this Sprite is relative only to the current **View**. The **sprite** argument is a pointer to a **SimpleSprite** data structure. (You must allocate an actual **SimpleSprite** structure for **sprite** to point to.) Set **newdata** to the address of an image data structure containing the new image. The data must reside in Chip (MEMF_CHIP) memory.

The structure for the new sprite image data is shown below. It is not a system structure, so it will not be found in the system includes, but it is described in the documentation for the **ChangeSprite()** call.

```
struct spriteimage
    {
    UWORD posctl[2];                /* position and control data for this Sprite */

    /* Two words per line of Sprite height, first of the two words contains the MSB for
     * color selection, second word contains LSB (colors 0,1,2,3 from allowable color
     * register selection set).   Color '0' for any Sprite pixel makes it transparent.
     */
    UWORD data[height][2];          /* actual Sprite image */

    UWORD reserved[2];              /* reserved, initialize to 0, 0 */
    };
```

Moving A Simple Sprite

MoveSprite() repositions a Simple Sprite. After this function is called, the Simple Sprite is moved to a new position relative to the upper left corner of the **ViewPort**. It is called as follows:

```
struct ViewPort     *vp;
struct SimpleSprite *sprite;
SHORT               x, y;

MoveSprite(vp, sprite, x, y);
```

There are three inputs to **MoveSprite()**. Set the **vp** argument to the address of the **ViewPort** with which this Simple Sprite interacts or 0 if this Simple Sprite's position is relative only to the current **View**. Set **sprite** to the address of your **SimpleSprite** data structure. The **x** and **y** arguments sepcify a pixel position to which the Simple Sprite is to be moved.

Relinquishing A Simple Sprite

The **FreeSprite()** function returns a hardware sprite allocated with **GetSprite()** to the system so that GELs or other tasks can use it. After you call **FreeSprite()**, the GELs system can use it to allocate VSprites. The syntax of this function is:

```
WORD sprite_number;

FreeSprite(sprite_number);
```

The **sprite_number** argument is the number (0-7) of the sprite to be returned to the system.

Controlling Sprite DMA

Two additional functions used with Simple Sprites are the graphics library macros **ON_SPRITE** and **OFF_SPRITE**. These macros can be used to control sprite DMA. **OFF_SPRITE** prevents the system from displaying any sprites, whether Simple Sprites or VSprites. **ON_SPRITE** restores the sprite display.

> *Be Careful With OFF_SPRITE.* The Intuition mouse pointer is a sprite. Thus, if OFF_SPRITE is used, Intuition's pointer will disappear too. Use care when calling OFF_SPRITE. The macro turns off sprite fetch DMA, so that no new sprite data is fetched. Whatever sprite data was last being displayed at this point will continue to be displayed for *every* line on the screen. This may lead to a vertical color bar if a sprite is being displayed when OFF_SPRITE is called.

Complete Simple Sprite Example

The following example demonstrates how to set up, move and free a Simple Sprite. The animtools.h file included is listed at the end of the chapter.

```
/* ssprite.c - Simple Sprite example
**
** SAS/C V5.10a
** lc -b1 -cfist -v -y ssprite.c
** blink FROM LIB:c.o ssprite.o LIB LIB:lc.lib LIB:amiga.lib TO ssprite
*/
#include <exec/types.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gfxmacros.h>
#include <graphics/sprite.h>
#include <intuition/intuitionbase.h>
#include <intuition/screens.h>
#include <hardware/custom.h>
#include <hardware/dmabits.h>
#include <libraries/dos.h>

#include <clib/graphics_protos.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/alib_stdio_protos.h>

#include <stdlib.h>

struct GfxBase *GfxBase = NULL;
struct IntuitionBase *IntuitionBase = NULL;
extern struct Custom far custom ;

/* real boring sprite data */
UWORD chip sprite_data[ ] = {
    0, 0,           /* position control          */
    0xffff, 0x0000, /* image data line 1, color 1 */
    0xffff, 0x0000, /* image data line 2, color 1 */
    0x0000, 0xffff, /* image data line 3, color 2 */
    0x0000, 0xffff, /* image data line 4, color 2 */
    0x0000, 0x0000, /* image data line 5, transparent */
    0x0000, 0xffff, /* image data line 6, color 2 */
    0x0000, 0xffff, /* image data line 7, color 2 */
    0xffff, 0xffff, /* image data line 8, color 3 */
    0xffff, 0xffff, /* image data line 9, color 3 */
    0, 0            /* reserved, must init to 0 0 */
    };

VOID main(int argc, char **argv)
{
struct SimpleSprite    sprite = {0};
struct ViewPort        *viewport;

WORD sprite_num;
SHORT delta_move, ktr1, ktr2, color_reg;
struct Screen *screen;
int return_code;

return_code = RETURN_OK;

if (NULL == (GfxBase = (struct GfxBase *) OpenLibrary("graphics.library",37L)))
    return_code = RETURN_FAIL;
else
    {
    if (NULL == (IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library",37L)))
        return_code = RETURN_FAIL;
    else
        {
        /* opened library, need a viewport to render a sprite over. */
        if (NULL == (screen = OpenScreenTagList(NULL, NULL)))
            return_code = RETURN_FAIL;
        else
            {
            viewport = &screen->ViewPort;
```

```
        if (-1 == (sprite_num = GetSprite(&sprite, 2)))
            return_code = RETURN_WARN;
        else
            {
            /* Calculate the correct base color register number, */
            /* set up the color registers.                       */
            color_reg = 16 + ((sprite_num & 0x06) << 1);
            printf("color_reg=%d\n", color_reg);
            SetRGB4(viewport, color_reg + 1, 12,  3,  8);
            SetRGB4(viewport, color_reg + 2, 13, 13, 13);
            SetRGB4(viewport, color_reg + 3,  4,  4, 15);

            sprite.x = 0;       /* initialize position and size info    */
            sprite.y = 0;       /* to match that shown in sprite_data   */
            sprite.height = 9;  /* so system knows layout of data later */

            /* install sprite data and move sprite to start position. */
            ChangeSprite(NULL, &sprite, (APTR)sprite_data);
            MoveSprite(NULL, &sprite, 30, 0);

            /* move the sprite back and forth. */
            for ( ktrl = 0, delta_move = 1;
                  ktrl < 6; ktrl++, delta_move = -delta_move)
                {
                for ( ktr2 = 0; ktr2 < 100; ktr2++)
                    {
                    MoveSprite( NULL, &sprite, (LONG)(sprite.x + delta_move),
                                (LONG)(sprite.y + delta_move) );
                    WaitTOF(); /* one move per video frame */

                    /* Show the effect of turning off sprite DMA. */
                    if (ktr2 == 40) OFF_SPRITE ;
                    if (ktr2 == 60) ON_SPRITE ;
                    }
                }
            /* NOTE:  if you turn off the sprite at the wrong time (when it
            ** is being displayed), the sprite will appear as a vertical bar
            ** on the screen.  To really get rid of the sprite, you must
            ** OFF_SPRITE while it is not displayed.  This is hard in a
            ** multi-tasking system (the solution is not addressed in
            ** this program).
            */
            ON_SPRITE ; /* just to be sure */
            FreeSprite((WORD)sprite_num);
            }
         (VOID) CloseScreen(screen);
         }
      CloseLibrary((struct Library *)IntuitionBase);
      }
   CloseLibrary((struct Library *)GfxBase);
   }
exit(return_code);
}
```

Using Virtual Sprites

This section describes how to set up the **VSprite** structure so that it represents a true VSprite. True VSprites are managed by the GELs system which converts them to Simple Sprites and displays them. (Later sections describe how a **VSprite** structure can be set up for Bobs and AnimComps.)

Before the system is told of a VSprite's existence, space for the **VSprite** data structure must be allocated and initialized to *correctly* represent a VSprite. Since the system does no validity checking on the **VSprite** structure, the result of using a bogus structure is usually a fireworks display, followed by a system failure.

The system software provides a way to detect collisions between VSprites and other on-screen objects. There is also a method of extending the **VSprite** structure to incorporate user defined variables. These subjects are applicable to all GELs and are explained later in "Collisions and GEL Structure Extensions".

SPECIFICATION OF VSPRITE STRUCTURE

The **VSprite** structure is defined in the include file <*graphics/gels.h*> as follows:

```
/* VSprite structure definition */
struct VSprite {
    struct VSprite *NextVSprite;
    struct VSprite *PrevVSprite;
    struct VSprite *DrawPath;
    struct VSprite *ClearPath;
    WORD            OldY, OldX;
    WORD            Flags;
    WORD            Y, X;
    WORD            Height;
    WORD            Width;
    WORD            Depth;
    WORD            MeMask;
    WORD            HitMask;
    WORD           *ImageData;
    WORD           *BorderLine;
    WORD           *CollMask;
    WORD           *SprColors;
    struct Bob     *VSBob;
    BYTE            PlanePick;
    BYTE            PlaneOnOff;
    VUserStuff      VUserExt;
    };
```

There are two primary ways to allocate and fill in space for **VSprite** data. They can be statically declared, or a memory allocation function can be called and they can be filled in programmatically. The declaration to statically set up a **VSprite** structure is listed below.

```
/* VSprite static data definition.
** must set the following for TRUE VSprites:
**    VSPRITE flag.
**    Width to 1.
**    Depth to 2.
**    VSBob to NULL.
*/
struct VSprite myVSprite =
    {
    NULL, NULL, NULL, NULL, 0, 0, VSPRITE, 0, 0, 5, 1, 2, 0, 0,
    &myImage, 0, 0, &mySpriteColors, NULL, 0x3, 0, 0
    };
```

This static allocation gives the required **VSprite** structure, but does not allocate or set up collision masks for the VSprite. Note that the **VSprite** structure itself does not need to reside in Chip memory.

Refer to the **makeVSprite()** and **freeVSprite()** functions in the animtools.c listing at the end of the chapter for an example of dynamically allocating, initializing and freeing a **VSprite** structure.

RESERVED VSPRITE MEMBERS

These **VSprite** structure members are reserved for system use (do not write to them):

NextVSprite and **PrevVSprite**	These are used as links in the GelsInfo list.
DrawPath and **ClearPath**	These are used for Bobs, not true VSprites.
OldY and **OldX**	Previous position holder, the system uses these for double buffered Bobs, but application programs can read them too.

The values can be set like this:

```
myVSprite.NextVSprite = NULL;
myVSprite.PrevVSprite = NULL;
myVSprite.DrawPath   = NULL;
myVSprite.ClearPath = NULL;
myVSprite.OldY = 0;
myVSprite.OldX = 0;
```

USING VSPRITE FLAGS

The **Flags** member of the **VSprite** structure is both read and written by the system. Some bits are used by the application to inform the system; others are used by the system to indicate things to the application.

The only **Flags** bits that are used by true VSprites are:

VSPRITE
: This may be set to indicate to the system that it should treat the structure as a true VSprite, not part of a Bob. This affects the interpretation of the data layout and the use of various system variables.

VSOVERFLOW
: The system sets this bit in the true VSprites that it is unable to display. This happens when there are too many in the same scan line, and the system has run out of Simple Sprites to assign. It indicates that this VSprite has not been displayed. If no sprites are reserved, this means that more than eight sprites touch one scan line. This bit will not be set for Bobs and should not be changed by the application.

GELGONE
: If the system has set GELGONE bit in the **Flags** member, then the GEL associated with this VSprite is not on the display at all, it is entirely outside the GEL boundaries. This area is defined by the **GelsInfo** members **topmost**, **bottommost**, **leftmost** and **rightmost** (see *<graphics/rastport.h>*). On the basis of that information, the application may decide that the object need no longer be part of the GEL list and may decide to remove it to speed up the consideration of other objects. Use **RemVSprite()** (or **RemBob()**, if it's a Bob) to do this. This bit should not be changed by the application.

The **VSprite.Flags** value should be initialized like this for a VSprite GEL:

```
myVSprite.Flags = VSPRITE;
```

VSPRITE POSITION

To control the position of a VSprite, the **x** and **y** variables in the **VSprite** structure are used. These specify where the upper left corner of the VSprite will be, relative to the upper left corner of the playfield area it appears over. So if VSprites are used under Intuition and within a screen, they will be positioned relative to the upper left-hand corner of the screen.

In a 320 by 200 screen, a **y** value of 0 puts the VSprite at the top of that display, a **y** value of (200 - VSprite height) puts the VSprite at the bottom. And an **x** value of 0 puts the VSprite at the left edge of that display, while an **x** value of (320 - VSprite width) puts the VSprite at the far right. Values of less than (0,0) or greater than (320, 200) may be used to move the VSprite partially or entirely off the screen, if desired.

See the "Graphics Primitives" chapter for more information on display coordinates and display size. See the *Amiga Hardware Reference Manual* for more information on hardware sprites.

> *Position VSprites Properly.* It is important that the starting position of true VSprites is not less than -20 in the y direction, which is the start of the active display area for sprites. Also, if they are moved too far to the left, true VSprites may not have enough DMA time to be displayed.

The x, y values may be set like this to put the VSprite in the upper-left:

```
myVSprite.Y = 0;
myVSprite.X = 0;
```

VSPRITE IMAGE SIZE

A true VSprite is always one word (16 pixels) wide and may be any number of lines high. It can be made to appear thinner by making some pixels transparent. Like Simple Sprites, VSprite pixels are always the size of a pixel in low-resolution mode (320x200); regardless of the resolution the display is set to. To specify how many lines make up the VSprite image, the **VSprite** structure member, **Height**, is used. VSprites always have a **Depth** of two, allowing for three colors. The values may be set like this:

```
myVSprite.Width  = 1;      /* ALWAYS 1 for true VSprites. */
myVSprite.Height = 5;      /* The example height. */
myVSprite.Depth  = 2;      /* ALWAYS 2 for true VSprites. */
```

VSPRITES AND COLLISION DETECTION

Some members of the **VSprite** data structure are used for special purposes such as collision detection, user extensions or for system extensions (such as Bobs and AnimComps). For most applications these fields are set to zero:

```
myVSprite.HitMask    = 0;  /* These are all used for collision detection */
myVSprite.MeMask     = 0;
myVSprite.BorderLine = 0;
myVSprite.CollMask   = 0;

myVSprite.VUserExt = 0;    /* Only use this for user extensions to VSprite */

myVSprite.VSBob = NULL;    /* Only Bobs and AnimComps need this */
```

The special uses of this fields are explained further in the sections that follow.

VSPRITE IMAGE DATA

The **ImageData** pointer of the **VSprite** structure must be initialized with the address of the first word of the image data array. The image data array must be in Chip memory. It takes two sequential 16-bit words to define each line of a VSprite. This means that the data area containing the VSprite image is always *Height x 2* (10 in the example case) words long.

A VSprite image is defined just like a real hardware sprite. The combination of bits in corresponding locations in the two data words that define each line select the color for that pixel. The first of the pair of words supplies the low-order bit of the color selector for that pixel; the second word supplies the high-order bit.

These binary values select colors as follows:

00 - selects "transparent"
01 - selects the first of three VSprite colors
10 - selects the second VSprite color
11 - selects the third VSprite color

In those areas where the combination of bits yields a value of 0, the VSprite is transparent. This means that the playfield, and all Bobs and AnimComps, and any VSprite whose priority is lower than this VSprite will all show through in transparent sections. For example:

(&VSprite->ImageData)	1010 0000 0000 0000
(&VSprite->ImageData + 1)	0110 0000 0000 0000

Reading from top to bottom, left to right, the combinations of these two sequential data words form the binary values of 01, 10, 11, and then all 00s. This VSprite's first pixel will be color 1, the next color 2, the third color 3. The rest will be transparent, making this VSprite appear to be three pixels wide. Thus, a three-color image, with some transparent areas, can be formed from a data set like the following sample:

Address	Binary Data	VSprite Image Data
mem	1111 1111 1111 1111	Defines top line
mem + 1	1111 1111 1111 1111	**3333 3333 3333 3333**
mem + 2	0011 1100 0011 1100	Defines second line
mem + 3	0011 0000 0000 1100	**0033 1100 0011 3300**
mem + 4	0000 1100 0011 0000	Defines third line
mem + 5	0000 1111 1111 0000	**0000 3322 2233 0000**
mem + 6	0000 0010 0100 0000	Defines fourth line
mem + 7	0000 0011 1100 0000	**0000 0032 2300 0000**
mem + 8	0000 0001 1000 0000	Defines fifth line
mem + 9	0000 0001 1000 0000	**0000 0003 3000 0000**

The **VSprite.Height** for this sample image is 5.

SPECIFYING THE COLORS OF A VSPRITE

The system software provides a great deal of versatility in the choice of colors for Virtual Sprites. Each VSprite has *its own set* of three colors, pointed to by **SprColors**, which the system jams into the display's Copper list as needed.

SprColors points to the first of three 16-bit values. The first value represents the color used for the VSprite bits that select color 1, the second value is color 2, and the third value is color 3. When the system assigns a hardware sprite to carry the VSprite's image, it jams these color values into the Copper list (the intermediate Copper list, *not* the color table), so that the **View**'s colors will be correct for this VSprite at the time the VSprite is displayed. It doesn't jam the original palette's colors back after the VSprite is done. If there is another VSprite later, that VSprite's colors will get jammed; if there is not another VSprite, the colors will remain the same until the next **ViewPort**'s colors get loaded.

If the **SprColors** pointer is set to NULL, that VSprite does not generate a color-change instruction stream for the Copper. Instead, the VSprite appears drawn in whatever color set that the hardware sprite happens to have in it already.

Since the registers are initially loaded with the colors from the **ViewPort's ColorMap**, if all VSprites have NULL **SprColors**, they will appear in the **ViewPort's** colors.

To continue our example, a set of colors can be declared and the VSprite colors set with the following statements:

```
WORD mySpriteColors[] = { 0x0000, 0x00f0, 0x0f00 }; /* Declare colors statically */

myVSprite.SprColors = mySpriteColors;             /* Assign colors to VSprite */
```

ADDING AND REMOVING VSPRITES

Once a true VSprite has been set up and initialized, the obvious next step is to give it to the system by adding it to the GEL list. The VSprite may then be manipulated as needed. Before the program ends, the VSprite should be removed from the GELs list by calling **RemVSprite()**. A typical calling sequence could be performed like so:

```
struct VSprite  myVSprite = {0};
struct RastPort myRastPort = {0};

AddVSprite(&myVSprite, &myRastPort);

/* Manipulate the VSprite as needed here */

RemVSprite(&myVSprite);
```

The **&myVSprite** argument is a fully initialized **VSprite** structure and **&myRastPort** is the **RastPort** with which this VSprite is to be associated. Note that you will probably not like the results if you try to **RemVSprite()** a VSprite that has not been added to the system with **AddVSprite()**. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for additional information on these functions.

CHANGING VSPRITES

Once the VSprite has been added to the GELs list and is in the display, some of its characteristics can be changed dynamically by:

- Changing **y**, **x** to a new VSprite position
- Changing **ImageData** to point to a new VSprite image
- Changing **SprColors** to point to a new VSprite color set

Study the next two sections to find out how to reserve hardware Sprites for use outside the VSprite system and how to assign the VSprites.

GETTING THE VSPRITE LIST IN ORDER

When the system has displayed the last line of a VSprite, it is able to reassign the hardware sprite to another VSprite located at a lower position on the screen. The system allocates hardware sprites in the order in which it encounters the VSprites in the list. Therefore, the list of VSprites must be sorted before the system can assign the use of the hardware Sprites correctly.

The function **SortGList()** must be used to get the GELs in the correct order before the system is asked to display them. *This sorting step is essential!* It should be done before calling **DrawGList()**, whenever a GEL has changed position. This function is called as follows:

```
struct RastPort myRastPort = {0};

SortGList(&myRastPort);
```

The only argument is a pointer to the **RastPort** structure containing the **GelsInfo**.

DISPLAYING THE VSPRITES

The next few sections explain how to display the VSprites. The following system functions are used:

DrawGList()	Draws the VSprites into the current **RastPort**.
MrgCop()	Installs the VSprites into the display.
LoadView()	Asks the system to display the new **View**.
WaitTOF()	Synchronizes the functions with the display.

Drawing the Graphics Elements

The system function called **DrawGList()** looks through the list of GELS and prepares the necessary Copper instructions and memory areas to display the data. This function is called as follows:

```
struct RastPort myRastPort = {0};
struct ViewPort myViewPort = {0};

DrawGList(&myRastPort, &myViewPort);
```

The **myRastPort** argument specifies the **RastPort** containing the **GelsInfo** list with the VSprites that you want to display. The **&myViewPort** argument is a pointer to the **ViewPort** for which the VSprites will be created.

Merging VSprite Instructions

Once **DrawGList()** has prepared the necessary instructions and memory areas to display the data, the VSprites are installed into the display with **MrgCop()**. (**DrawGList()** does not actually draw the VSprites, it only prepares the Copper instructions.)

```
struct View *view;

MrgCop(view);
```

The **view** is a pointer to the **View** structure whose Copper instructions are to be merged.

Loading the New View

Now that the display instructions include the definition of the VSprites, the system can display this newly configured **View** with the **LoadView()** function:

```
struct View *view;

LoadView(view);
```

Again, **view** is a pointer to the **View** that contains the the new Copper instruction list (if you are using GELs in an Intuition Screen, do not call **LoadView()**.)

The Copper instruction lists are double-buffered, so this instruction does not actually take effect until the next display field occurs. This avoids the possibility of some function trying to update the Copper instruction list while the Copper is trying to use it to create the display.

Synchronizing with the Display

To synchronize application functions with the display, call the system function **WaitTOF()**. **WaitTOF()** holds your task until the vertical-blanking interval (blank area at the top of the screen) has begun. At that time, the system has retrieved the current Copper instruction list and is ready to allow generation of a new list.

```
WaitTOF();
```

WaitTOF() takes no arguments and returns no values. It simply suspends your task until the video beam is at the top of field.

Complete VSprite Example

The listing given here shows a complete VSprite example. This program requires the animtools.c, animtools.h and animtools_proto.h support files in order to compile and run. These files are listed at the end of this chapter.

```
/* vsprite.c
**
** SAS/C V5.10a
** lc -b1 -cfist -v -y vsprite.c
** blink FROM LIB:c.o vsprite.o animtools.o LIB LIB:lc.lib LIB:amiga.lib TO vsprite
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuitionbase.h>
#include <graphics/gfx.h>
#include <graphics/gfxbase.h>
#include <graphics/gels.h>
#include <graphics/collide.h>
#include <libraries/dos.h>
#include <stdlib.h>
#include "animtools.h"

VOID borderCheck(struct VSprite *hitVSprite, LONG borderflags);
VOID process_window(struct Window *win, struct RastPort *myRPort, struct VSprite *MyVSprite);
VOID do_VSprite(struct Window *win, struct RastPort *myRPort);
VOID vspriteDrawGList(struct Window *win, struct RastPort *myRPort);

struct GfxBase         *GfxBase;     /* pointer to Graphics library  */
struct IntuitionBase *IntuitionBase; /* pointer to Intuition library */
```

```
int return_code;
#define GEL_SIZE  4 /* number of lines in the vsprite */

/* VSprite data - there are two sets that are alternated between. */
/* note that this data is always displayed as low resolution.      */
WORD chip vsprite_data1[] = { 0x7ffe, 0x80ff,
                              0x7c3e, 0x803f,
                              0x7c3e, 0x803f,
                              0x7ffe, 0x80ff,
                              0, 0 };

WORD chip vsprite_data2[] = { 0x7ffe, 0xff01,
                              0x7c3e, 0xfc01,
                              0x7c3e, 0xfc01,
                              0x7ffe, 0xff01,
                              0, 0 };

WORD mySpriteColors[] =     { 0x0000, 0x00f0, 0x0f00 };
WORD mySpriteAltColors[] =  { 0x000f, 0x0f00, 0x0ff0 };

NEWVSPRITE myNewVSprite = {             /* information for the new VSprite       */
        /* Image data, sprite color array word width (must be 1 for true VSprite) */
        vsprite_data1, mySpriteColors,1,
        /* Line height, image depth (must be 2 for true VSprite), x, y position    */
        GEL_SIZE, 2, 160, 100,
        /* Flags (VSPRITE == true VSprite), hit mask and me mask                  */
        VSPRITE, 1 << BORDERHIT, 0
        };

struct NewWindow myNewWindow = {        /* information for the new window */
    80, 20, 400, 150, -1, -1, CLOSEWINDOW | INTUITICKS,
    ACTIVATE | WINDOWCLOSE | WINDOWDEPTH | RMBTRAP | WINDOWDRAG,
    NULL, NULL, "VSprite", NULL, NULL, 0, 0, 0, 0, WBENCHSCREEN
    };

/* Basic VSprite display subroutine */
VOID vspriteDrawGList(struct Window *win, struct RastPort *myRPort)
{
SortGList(myRPort);
DrawGList(myRPort, ViewPortAddress(win));
RethinkDisplay();
}

/* Collision routine for vsprite hitting border.  Note that when the collision is VSprite to */
/* VSprite (or Bob to Bob, Bob to AnimOb, etc), then the parameters are both pointers to a VSprite. */
VOID borderCheck(struct VSprite *hitVSprite, LONG borderflags)
{
if (borderflags & RIGHTHIT)
    {
    hitVSprite->SprColors = mySpriteAltColors;
    hitVSprite->VUserExt  = -40;
    }
if (borderflags & LEFTHIT)
    {
    hitVSprite->SprColors = mySpriteColors;
    hitVSprite->VUserExt  = 20;
    }
}

/* Process window and dynamically change vsprite. Get messages. Go away on          */
/* CLOSEWINDOW.  Update and redisplay vsprite on INTUITICKS. Wait for more messages. */
VOID process_window(struct Window *win, struct RastPort *myRPort, struct VSprite *myVSprite)
{
struct IntuiMessage *msg;

FOREVER
    {
    Wait(1L << win->UserPort->mp_SigBit);
    while (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
        {
        /* Only CLOSEWINDOW and INTUITICKS are active */
        if (msg->Class == CLOSEWINDOW)
            {
            ReplyMsg((struct Message *)msg);
            return;
            }
```

```
        /* Must be an INTUITICKS:  change x and y values on the fly.  Note offset by
        ** window left and top edge--sprite relative to the screen, not window.  Divide
        ** the MouseY in half to adjust for Lores movement increments on a Hires screen.
        */
        myVSprite->X = win->LeftEdge + msg->MouseX + myVSprite->VUserExt;
        myVSprite->Y = win->TopEdge  + msg->MouseY/2 + 1;
        ReplyMsg((struct Message *)msg);
        }
    /* Got a message, change image data on the fly */
    myVSprite->ImageData = (myVSprite->ImageData == vsprite_data1) ? vsprite_data2 : vsprite_data1;
    SortGList(myRPort);
    DoCollision(myRPort);
    vspriteDrawGList(win, myRPort);
    }
}

/* Working with the VSprite.  Setup the GEL system and get a new VSprite (makeVSprite()).   */
/* Add VSprite to the system and display.  Use the vsprite.  When done, remove VSprite and  */
/* update the display without the VSprite.  Cleanup everything.                             */
VOID do_VSprite(struct Window *win, struct RastPort *myRPort)
{
struct VSprite       *myVSprite;
struct GelsInfo       *my_ginfo;

if (NULL == (my_ginfo = setupGelSys(myRPort, 0xfc)))
    return_code = RETURN_WARN;
else
    {
    if (NULL == (myVSprite = makeVSprite(&myNewVSprite)))
        return_code = RETURN_WARN;
    else
        {
        AddVSprite(myVSprite, myRPort);
        vspriteDrawGList(win, myRPort);
        myVSprite->VUserExt = 20;
        SetCollision(BORDERHIT, borderCheck, myRPort->GelsInfo);
        process_window(win, myRPort, myVSprite);
        RemVSprite(myVSprite);
        freeVSprite(myVSprite);
        }
    vspriteDrawGList(win, myRPort);
    cleanupGelSys(my_ginfo, myRPort);
    }
}

/* Example VSprite program.  First open up the libraries and a window. */
VOID main(int argc, char **argv)
{
struct Window       *win;
struct RastPort    myRPort = {0};

return_code = RETURN_OK;

if (NULL == (GfxBase = (struct GfxBase *)OpenLibrary(GRAPHICSNAME,37L)))
    return_code = RETURN_FAIL;
else
    {
    if (NULL == (IntuitionBase = (struct IntuitionBase *)OpenLibrary(INTUITIONNAME,37L)))
        return_code = RETURN_FAIL;
    else
        {
        if (NULL == (win = OpenWindow(&myNewWindow)))
            return_code = RETURN_WARN;
        else
            {
            InitRastPort(&myRPort);
            myRPort = win->WScreen->RastPort;       /* Copy the structure. */
            do_VSprite(win, &myRPort);
            CloseWindow(win);
            }
        CloseLibrary((struct Library *)IntuitionBase);
        }
    CloseLibrary((struct Library *)GfxBase);.
    }
exit(return_code);
}
```

VSprite Advanced Topics

This section describes advanced topics pertaining to VSprites. It contains details about reserving hardware sprites for use outside of the GELs VSprite system, information about how VSprites are assigned, and more information about VSprite colors.

Reserving Hardware Sprites

To prevent the VSprite system from using specific hardware sprites, set the **sprRsrvd** member of the **GelsInfo** structure. The pointer to the **GelsInfo** structure is contained in the **RastPort** structure. If all of the bits of this 8-bit value are ones (0xFF), then all of the hardware sprites may be used by the VSprite system. If any of the bits is a 0, the sprite corresponding to that bit will not be utilized by VSprites.

> *Reserving Can Cause Problems.* Reserving sprites increases the likelihood of the system not being able to display a VSprite (VSOVERFLOW). See the next section, "How VSprites are Assigned," for further details on this topic.

You reserve a sprite by setting its corresponding bit in **sprRsrvd**. For instance, to reserve sprite zero only, set sprRsrvd to 0x01. To reserve sprite three only, set **sprRsrvd** to 0x08.

If a hardware sprite is reserved, the system will not consider it when it makes VSprite assignments. Remember, hardware sprite pairs share color register sets. If a hardware sprite is reserved, its mate should probably be reserved too, otherwise the reserved sprite's colors will change as the unreserved mate is assigned different VSprites. For example, it is common practice to reserve Sprites 0 and 1, so that the Intuition pointer (Sprite 0) is left alone. This could be accomplished with the following statements:

```
struct RastPort myRastPort = {0};   /* the View structure is defined */

myRastPort.GelsInfo->sprRsrvd = 0x03;   /* reserve 0 and 1 */
```

The **GfxBase** structure may be examined to find which sprites are already in use. This may, at your option, impact what sprites you reserve. If Intuition is running, sprite 0 will already be in use as its pointer.

The reserved sprite status is accessible as

```
currentreserved = GfxBase->SpriteReserved;
```

The next section presents a few trouble-shooting techniques for VSprite assignment.

How VSprites Are Assigned

Although VSprites are managed for you by the GELs system there are some underlying limitations which could cause the system to run out of VSprites.

As the system goes through the GEL list during **DrawGList()**, whenever it finds a true VSprite, it goes through the following procedure. If there is a Simple Sprite available (after the reserved sprites and preceding VSprites are accounted for), Copper instructions are added that will load the sprite hardware with this VSprite's data at the right point on the screen. It may need to add a Copper instruction sequence to load the display's colors associated with the sprite as well.

There are only 8 *real* sprite DMA channels. The system will run out of hardware sprites if it is asked to display more than eight VSprites on one scan line. This limit goes down to four when the VSprites have different **SprColor** pointers. During the time that there is a conflict, the VSprites that could not be put into Simple Sprites will disappear. They will reappear when (as the VSprites are moved about the screen) circumstances permit.

These problems can be alleviated by taking some precautions:

- Minimize the number of VSprites to appear on a single horizontal line.

- If colors for some Virtual Sprites are the same, make sure that the pointer for each of the **VSprite** structures for these Virtual Sprites points to the same memory location, rather than to a duplicate set of colors elsewhere in memory. The system will know to map these into Sprite pairs.

If a VSprite's **SprColors** are set to NULL, the VSprite will appear in the **ViewPort**'s **ColorMap** colors. The system will display the VSprite in any one of a set of four different possible color groupings as indicated in the Simple Sprite section above.

If **SprColors** points to a color set, the system will jam **SprColors** into the display hardware (via the Copper list), effectively overriding those **ColorMap** registers. The values in the **ColorMap** are *not* overwritten, but anything in the background display that used to appear in the **ColorMap** colors will appear in **SprColors** colors.

How VSprite and Playfield Colors Interact

At the start of each display, the system loads the colors from the **ViewPort**'s color table into the display's hardware registers, so whatever is rendered into the **BitMap** is displayed correctly. But if the VSprite system is used, and the colors are specified (via **SprColors**) for each VSprite, the **SprColors** will be loaded by the system into the display hardware, as needed. The system does this by generating Copper instructions that will jam the colors into the hardware at specific moments in the display cycle. Any **BitMap** rendering, including Bobs, which share colors with VSprites, may change colors constantly as the video display beam progresses down the screen.

This color changing can be avoided by taking one of the following precautions:

- Use a four bitplane playfield, which only allows the lower 16 colors to be rendered into the **BitMap** (and allows Hires display mode).

- If a 32-color playfield display is being used, avoid rendering in colors 17-19, 21-23, 25-27, and 29-32, which are the colors affected by the VSprite system.

- Specify the VSprite **SprColors** pointer as a value of NULL to avoid changing the contents of any of the hardware sprite color registers. This may cause the VSprites to change colors depending on their positions relative to each other, as described in the previous section.

Using Bobs

The following section describes how to define a Bob (blitter object). Like VSprites, a Bob is a software construct designed to make animation easier. The main advantage of a Bob over a VSprite is that it allows more colors and a width greater than 16 pixels to be defined.

To create a Bob, you need both a **Bob** structure and a **VSprite** structure. The components common to all GELs — height, collision-handling information, position in the drawing area and pointers to the image definition — are part of the **VSprite** structure. The added features — such as drawing sequence, data about saving and restoring the background, and other features not applicable to VSprites — are further specified in the **Bob** structure.

THE VSPRITE STRUCTURE AND BOBS

The root **VSprite** structure is set up as described earlier for true VSprites, with the following exceptions:

Y, X	Bob position is always in pixels that are the same resolution as the display.
Flags	For Bobs, the VSPRITE flag must be cleared. SAVEBACK or OVERLAY can also be used.
Height, Width	Bob pixels are the size of the background pixels. The **Width** of Bobs may be greater than one word.
Depth	The **Depth** of a Bob may be up to as deep as the playfield, provided that enough image data is provided.
ImageData	This is still a pointer to the image, but the data there is organized differently.
SprColors	This pointer should be set to NULL for Bobs.
VSBob	This is a pointer to the **Bob** structure set up as described below.

VSPRITE FLAGS AND BOBS

The bits in the **VSprite.Flags** field that apply to a Bob are the VSPRITE flag, the SAVEBACK flag and the OVERLAY flag. When a **VSprite** structure is used to define a Bob, the VSPRITE flag in the **VSprite.Flags** field must be set to zero. This tells the system that this GEL is a Bob type.

To have the GEL routines save the background before the Bob is drawn and restore the background after the Bob is removed, specify the SAVEBACK flag (stands for ''save the background'') in the **VSprite** structure **Flags** field. If this flag is set, the **SaveBuffer** must have been allocated, which is where the system puts this saved background area. The buffer must be large enough to save all the background bitplanes, regardless of how many planes the Bob has. The size in words can be calculated as follows:

```
/* Note that Bob.Width is in units of words. */
size = Bob.Width * Bob.Height * RastPort.BitMap.Depth;
```

To allocate this space, the graphics function **AllocRaster()** can be used. **AllocRaster()** takes the width in bits, so it is a convenient way to allocate the space needed. The **makeBob()** routine below shows another way to correctly allocate this buffer. For example:

```
/* space for 16 bits times 5 lines times 5 bitplanes */
myBob.SaveBuffer = AllocRaster( (UWORD) 16, (UWORD) (5 * 5) );
```

Warning: The **SaveBuffer** must be allocated from Chip memory and contain an even number of word-aligned bytes. The **AllocRaster()** function does this for you. The **AllocRaster()** function rounds the width value up to the next integer multiple of 16 bits which is greater than or equal to the current value an it obtains memory from the Chip memory pool.

OVERLAY is the other **VSprite.Flags** item that applies to Bobs. If this flag is set, it means that the background's original pixels show through in any area where there are 0 bits in the Bob's shadow mask (**ImageShadow**, explained later). The space for the **ImageShadow** shadow mask must have been allocated and initialized. The **ImageShadow** mask must be allocated from Chip memory.

If the OVERLAY bit is cleared, the system uses the *entire rectangle of words* that define the Bob image to *replace* the playfield area at the specified x,y coordinates. See the paragraphs below called "ImageShadow."

THE BOB STRUCTURE

The **Bob** structure is defined in the include file *<graphics/gels.h>* as follows:

```
struct Bob
    {
    WORD              Flags;     /* general purpose flags              */
    WORD             *SaveBuffer;/* buffer for background save        */
    WORD             *ImageShadow; /* shadow mask of image             */
    struct Bob       *Before;    /* draw this Bob before Bobs on this list */
    struct Bob       *After;     /* draw this Bob after Bobs on this list  */
    struct VSprite   *BobVSprite;/* this Bob's VSprite definition      */
    struct AnimComp  *BobComp;   /* pointer to this Bob's AnimComp def */
    struct DBufPacket *DBuffer;  /* pointer to this Bob's dBuf packet  */
    BUserStuff        BUserExt;  /* Bob user extension                 */
    };
```

The **Bob** structure itself does not need to be in Chip memory. The (global) static declaration of a **Bob** structure could be done like so:

```
struct Bob myBob =
    {
    0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0
    };
```

However, since most of the **Bob** structure members are pointers, it is more common to allocate and set the **Bob** up dynamically. Refer to the **makeBob()** and **freeBob()** functions in the "animtools.c" example at the end of the chapter for an example of allocating, initializing and freeing a **Bob** structure.

Linking Bob and VSprite Structures

The **VSprite** and **Bob** structures must point to one another, so that the system can find the entire GEL. The structures are linked with statements like this:

```
myBob.BobVSprite = &myVSprite;
myVSprite.VSBob  = &myBob;
```

Now the system (and the application program) can go back and forth between the two structures to obtain the various **Bob** variables.

USING BOB FLAGS

The following paragraphs describe how to set the **Flags** field in the **Bob** structure (note that these flags do not apply to the **Flags** field of the **VSprite** structure).

SAVEBOB

To tell the system not to erase the old image of the Bob when the Bob is moved, specify the SAVEBOB flag in the **Bob** structure **Flags** field. This makes the Bob behave like a paintbrush. It has the opposite effect of SAVEBACK.

It's Faster To Draw A New Bob. It takes longer to preserve and restore the raster image than simply to draw a new Bob image wherever required.

BOBISCOMP

If this Bob is part of an **AnimComp**, set the BOBISCOMP flag in the **Bob** structure to 1. If the flag is a 1, the pointer named **BobComp** must have been initialized. Otherwise, the system ignores the pointer, and it may be left alone (though it's good practice to initialize it to NULL). See ''Animation Structures and Controls'' for a discussion of AnimComps.

BWAITING

This flag is used solely by the system, and should be left alone. When a Bob is waiting to be drawn, the system sets the BWAITING flag in the **Bob** structure to 1. This occurs only if the system has found a **Before** pointer in this **Bob**'s structure that points to another Bob. Thus, the system flag BWAITING provides current draw-status to the system. Currently, the system clears this flag on return from each call to **DrawGList()**.

BDRAWN

This is a system status flag that indicates to the system whether or not this Bob has already been drawn. Therefore, in the process of examining the various **Before** and **After** flags, the drawing routines can determine the drawing sequence. The system clears this flag on return from each call to **DrawGList()**.

BOBSAWAY

To initiate the removal of a Bob during the next call to **DrawGList()**, set BOBSAWAY to 1. Either the application or the system may set this **Bob** structure system flag. The system restores the background where it has last drawn the Bob. The system will unlink the Bob from the system GEL list the next time **DrawGList()** is called, unless the application is using double-buffering. In that case, the Bob will not be unlinked and completely removed until two calls to **DrawGList()** have occurred and the Bob has been removed from both buffers. The **RemBob()** macro sets the BOBSAWAY flag.

BOBNIX

When a Bob has been completely removed, the system sets the BOBNIX flag to 1 on return from **DrawGList()**. In other words, when the background area has been fully restored and the Bob has been removed from the GEL list, this flag in is set to a 1. BOBNIX is especially significant when double-buffering because when an application asks for a Bob to be removed, the system must remove it from both the drawing buffer *and* from the display buffer. Once BOBNIX has been set, it means the Bob has been removed from both buffers and the application is free to reuse or deallocate the Bob.

SAVEPRESERVE

The SAVEPRESERVE flag is a double-buffer version of the SAVEBACK flag. If using double-buffering and wishing to save and restore the background, set SAVEBACK to 1. SAVEPRESERVE is used by the system to indicate whether the Bob in the ''other'' buffer has been restored; it is for system use only.

SPECIFYING THE SIZE OF A BOB

Bobs do not have the 16-pixel width limit that applies to VSprites. To specify the overall size of a Bob, use the **Height** and **Width** members of the root **VSprite** structure. Specify the **Width** as the number of 16-bit words it takes to fully contain the object. The number of lines is still specified with the **Height** member in the **VSprite** data structure.

As an example, suppose the Bob is 24 pixels wide and 20 lines tall. Use statements like the following to specify the size:

```
myVSprite.Height = 20;  /* 20 lines tall. */
myVSprite.Width  = 2;   /* 24 bits fit into two words. */
```

Because Bobs are drawn into the background playfield, the pixels of the Bob are the same size as the background pixels, and share the color palette of the **ViewPort**.

SPECIFYING THE SHAPE OF A BOB

The layout of the data of a Bob's image is different from that of a VSprite because of the way the system retrieves data to draw Bobs. VSprite images are organized in a way convenient to the Sprite hardware; Bob images are set up for easy blitter manipulation. The **ImageData** pointer is still initialized to point to the first word of the image definition.

> *Note:* As with all image data, a Bob's **ImageData** must be in Chip memory for access by the blitter.

The sample image below shows the same image defined as a VSprite in the "Using Virtual Sprites" section above. The data here, however, is laid out for a Bob. The shape is 2 planes deep and is triangular:

	<first bitplane data>	
mem	1111 1111 1111 1111	Least significant bit of sprite line 1
mem + 1	0011 1100 0011 1100	Least significant bit of sprite line 2
mem + 2	0000 1100 0011 0000	Least significant bit of sprite line 3
mem + 3	0000 0010 0100 0000	Least significant bit of sprite line 4
mem + 4	0000 0001 1000 0000	Least significant bit of sprite line 5
	<second bitplane data>	
mem + 5	1111 1111 1111 1111	Most significant bit of sprite line 1
mem + 6	0011 0000 0000 1100	Most significant bit of sprite line 2
mem + 7	0000 1111 1111 0000	Most significant bit of sprite line 3
mem + 8	0000 0011 1100 0000	Most significant bit of sprite line 4
mem + 9	0000 0001 1000 0000	Most significant bit of sprite line 5
	<more bitplanes of data if Bob is deeper>	

SPECIFYING THE COLORS OF A BOB

Typically a five-bitplane, low-resolution mode display allows playfield pixels (and therefore, Bob pixels) to be selected from any of 32 active colors out of a system palette of 4,096 different color choices. Bob colors are limited to the colors used in the background playfield.

The system ignores the **sprColors** member of the **VSprite** structure when the **VSprite** structure is the root of a Bob. Instead, the Bob's colors are determined by the combination of the **Depth** of the Bob image and its **PlanePick, PlaneOnOff** and **ImageShadow** members.

Use the **Depth** member in the **VSprite** structure to indicate how many planes of image data is provided to define the Bob. This also defines how many colors the Bob will have. The combination of bits in corresponding Y,X positions in each bitplane determines the color of the pixel at that position.

For example, if a **Depth** of one plane is specified, then the bits of that image allow only two colors to be selected: one color for each bit that is a 0, a second color for each bit that is a 1. Likewise, if there are 5 planes of image data, all 32 colors can be used in the Bob. The Bob **Depth** must not exceed the background depth. Specify **Depth** using a statement such as the following:

```
myVSprite.Depth = 5;    /* Allow a 32 color, 5-bitplane image. */
```

OTHER ITEMS INFLUENCING BOB COLORS

The three other members in the **VSprite** structure that affect the color of Bob pixels are **ImageShadow**, **PlanePick**, and **PlaneOnOff**.

ImageShadow

The **ImageShadow** member is a pointer to the shadow mask of a Bob. A shadow mask is the logical *or* of all bitplanes of a Bob image. The system uses the shadow mask in conjunction with **PlaneOnOff**, discussed below, for color selection. It also uses the shadow mask to ''cookie cut'' the bits that will be overwritten by this Bob, to save and later restore the background.

The following figure shows the shadow mask of the image described above.

mem + 0	1111 1111 1111 1111	Shadow mask for line 1
mem + 1	0011 1100 0011 1100	Shadow mask for line 2
mem + 2	0000 1111 1111 0000	Shadow mask for line 3
mem + 3	0000 0011 1100 0000	Shadow mask for line 4
mem + 4	0000 0001 1000 0000	Shadow mask for line 5

Space for the **ImageShadow** must be provided and this pointer initialized to point to it. The amount of memory needed is equivalent to one plane of the image:

```
shadow_size = myBob->BobVSprite->Height * myBob->BobVSprite->Width;
```

The example image is 5 high and 1 word wide, so, 5 words must be made available.

Note: The **ImageShadow** memory must be allocated from Chip memory (MEMF_CHIP).

PlanePick

Because the **Depth** of the Bob can be less than the background, the **PlanePick** member is provided so that the application can indicate which background bitplanes are to have image data put into them. The system starts with the least significant plane of the Bob, and scans **PlanePick** starting at the least significant bit, looking for a plane of the **RastPort** to put it in.

For example, if **PlanePick** has a binary value of: 0 0 0 0 0 0 1 1 (0x03) then the system draws the first plane of the Bob's image into background plane 0 and the second plane into background plane 1.

Alternatively, a **PlanePick** value of: 0 0 0 1 0 0 1 0 (0x12) directs the system to put the first Bob plane into plane 1, and the second Bob plane into plane 4.

PlaneOnOff

What happens to the background planes that aren't picked? The shadow mask is used to either set or clear the bits in those planes in the exact shape of the Bob if OVERLAY is set, otherwise the entire rectangle containing the Bob is used. The **PlaneOnOff** member tells the system whether to put down the shadow mask as zeros or ones for each plane. The relationship between bit positions in **PlaneOnOff** and background plane numbers is identical to **PlanePick**: the least significant bit position indicates the lowest-numbered bitplane. A zero bit clears the shadow mask shape in the corresponding plane, while a one bit sets the shadow mask shape. The planes Picked by **PlanePick** have image data — not shadow mask — blitted in.

This provides a great deal of color versatility. One image definition can be used for many Bobs. By having different **PlanePick / PlaneOnOff** combinations, each Bob can use a different subset of the background color set.

There is a member in the **VSprite** structure called **CollMask** (the collision mask, covered under "Detecting GEL Collisions") for which the application may also reserve some memory space. The **ImageShadow** and **CollMask** pointers usually, but not necessarily, point to the same data, which must be located in Chip memory. If they point to the same location, obviously, the memory only need be allocated once.

An example of the kinds of statements that accomplish these actions (see the **makeVSprite()** and **makeBob()** examples for more details):

```
#define BOBW 1
#define BOBH 5
#define BOBD 2

/* Data definition from example layout */
WORD chip BobData[]=
    {
    0xFFFF, 0x300C, 0x0FF0, 0x03C0, 0x0180,
    0xFFFF, 0x3E7C, 0x0C30, 0x03C0, 0x0180
    };

/* Reserve space for the collision mask for this Bob */
WORD chip BobCollision[BOBW * BOBH];

myVSprite.Width  = BOBW;     /* Image is 16 pixels wide (1 word) */
myVSprite.Height = BOBH;    /* 5 lines for each plane of the Bob */
myVSprite.Depth  = BOBD;     /* 2 Planes are in ImageData */

/* Show the system where it can find the data image of the Bob */
myVSprite.ImageData = BobData;
```

```
/* binary 0101, render image data into bitplanes 0 and 2 */
myVSprite.PlanePick = 0x05;

/* binary 0000, means colors 1, 4, and 5 will be used.
*  binary 0010 would mean colors 3, 6, and 7.
*    "    1000   "    "     "    9, C, and D.
*    "    1010   "    "     "    B, E, and F.
*/
myVSprite.PlaneOnOff = 0x00;

/* Where to put collision mask */
myVSprite.CollMask = BobCollision;

/* Tell the system where it can assemble a GEL shadow */
/* Point to same area as CollMask */
myBob.ImageShadow = BobCollision;

/* Create the Sprite collision mask in the VSprite structure */
InitMasks(&myVSprite);
```

BOB PRIORITIES

This subsection describes the choices for inter-Bob priorities. The inter-Bob priorities tell the system what order to render the Bobs. Bobs rendered earlier will appear to be behind later Bobs. A Bob drawn earlier is said to have the lower priority and a Bob drawn later is said to have the higher priority. Thus, the highest priority Bob will be drawn last and will never be obstructed by another Bob.

Letting the System Decide Priorities

The priority issue can be ignored and the system will render the Bobs as it finds them in the **GelsInfo** list. To do this, set the Bob's **Before** and **After** pointers to NULL. Since the **GelsInfo** list is sorted by GEL x, y values, Bobs that are higher on the display will appear behind the lower ones, and Bobs that are more to the left on the display will appear behind Bobs on the right.

As Bobs are moved about the display, their priorities will change.

Specifying the Drawing Order

To specify the priorities of the Bobs, use the **Before** and **After** pointers. **Before** points to the Bob that this Bob should be drawn before, and **After** points to the Bob that this Bob should be drawn after. By following these pointers, from Bob to Bob, the system can determine the order in which the Bobs should be drawn. (Take care to avoid circular dependencies in this list!)

> *Note:* This terminology is often confusing, but, due to historical reasons, cannot be changed. The system does *not* draw the Bobs on the **Before** list first, it draws the Bobs on the **After** list first. Next, it draws the current Bob, and, finally, the Bobs on the **Before** list.

For example, to assure that myBob1 always appears in front of myBob2, The **Before** and **After** pointers must be initialized so that the system will always draw myBob1 after myBob2.

```
myBob2.Before = &myBob1;     /* draw Bob2 before drawing Bob1 */
myBob2.After  = NULL;        /* draw Bob2 after  no other Bob */
myBob1.After  = &myBob2;     /* draw Bob1 after  drawing Bob2 */
myBob1.Before = NULL;        /* draw Bob1 before no other Bob */
```

As the system goes through the **GelsInfo** list, it checks the Bob's **After** pointer. If this is not NULL, it follows the **After** pointer until it hits a NULL. Then it starts rendering the Bobs, going back up the **Before** pointers until it hits a NULL. Then it continues through the **GelsInfo** list. So, it is important that *all* **Before** and **After** pointers of a group properly point to each other.

> *Note:* In a screen with a number of complex GELs, you may want to specify the **Before** and **After** order for Bobs that are not in the same AnimOb. This will keep large objects together. If you do not do this, you may have an object drawn with half of its Bobs in front of another object! Also, in sequences you only set the **Before** and **After** pointers for the active AnimComp in the sequence.

ADDING A BOB

To add a Bob to the system GEL list, use the **AddBob()** routine. The **Bob** and **VSprite** structures must be correct and cohesive when this call is made. See the **makeBob()** and **makeVSprite()** routines in the animtools.c file listed at the end of this chapter for a detailed example of setting up Bobs and VSprites. See the **setupGelSys()** function for a more complete example of the initialization of the GELs system.

For example:

```
struct GelsInfo myGelsInfo = {0};
struct VSprite dummySpriteA = {0}, dummySpriteB = {0};
struct Bob myBob = {0};
struct RastPort rastport = {0};

/* Done ONCE, for this GelsInfo.  See setupGelSys() at the end of this
** chapter for a more complete initialization of the Gel system
*/
InitGels(&dummySpriteA, &dummySpriteB, &myGelsInfo);

/* Initialize the Bob members here, then AddBob() */
AddBob(&myBob, &rastport);
```

REMOVING A BOB

Two methods may be used to remove a Bob. The first method uses the **RemBob()** macro. **RemBob()** causes the system to remove the Bob during the next call to **DrawGList()** (or two calls to **DrawGList()** if the system is double-buffered). **RemBob()** asks the system to remove the Bob at the next convenient time. See the description of the BOBSAWAY and BOBNIX flags above. It is called as follows:

```
struct Bob myBob = {0};

RemBob(&myBob);
```

The second method uses the **RemIBob()** routine. **RemIBob()** tells the system to remove this Bob immediately. For example:

```
struct Bob      myBob = {0};
struct RastPort rastport = {0};
struct ViewPort viewport = {0};

RemIBob(&myBob, &rastport, &viewport);
```

This causes the system to erase the Bob from the drawing area and causes the immediate erasure of any other Bob that had been drawn subsequent to (and on top of) this one. The system then unlinks the Bob from the system GEL list. To redraw the Bobs that were drawn on top of the one just removed, another call to **DrawGList()** must be made.

SORTING AND DISPLAYING BOBS

As with VSprites, the **GelsInfo** list must be sorted before any Bobs can be displayed. This is accomplished with the **SortGList()** function. For Bobs, the system uses the position information to decide inter-Bob priorities, if not explicitly set by using the **Bob.Before** and **Bob.After** pointers.

Once the **GelsInfo** list has been sorted, the Bobs in the list can be displayed by calling **DrawGList()**. This call should then be followed by a call to **WaitTOF()** if the application wants to be sure that the Bobs are rendered before proceeding. Call these functions as follows:

```
struct RastPort myRastPort = {0}; /* Of course, these have to be initialized... */
struct ViewPort myViewPort = {0};

SortGList(&myRastPort);
DrawGList(&myRastPort, &myViewPort);     /* Draw the elements (Bobs only) */
WaitTOF();
```

> *Warning:* If your **GelsInfo** list contains VSprites in addition to Bobs, you must also call **MrgCop()** and **LoadView()** to make all the GELs visible. Or, under Intuition, **RethinkDisplay()** must be called to make all the GELs visible.

CHANGING BOBS

The following characteristics of Bobs can be changed dynamically between calls to **DrawGList()**:

- To change the location of the Bob in the **RastPort** drawing area, adjust the **X** and **Y** values in the **VSprite** structure associated with this Bob.
- To change a Bob's appearance, the pointer to the **ImageData** in the associated **VSprite** structure may be changed. Note that a change in the **ImageData** also requires a change or recalculation of the **ImageShadow**, using **InitMasks()**.
- To change a Bob's colors modify the **PlanePick**, **PlaneOnOff** or **Depth** parameters in the **VSprite** structure associated with this Bob.
- To change a Bob's display priorities, alter the **Before** and **After** pointers in the **Bob** structure.
- To change the Bob into a paintbrush, specify the SAVEBOB flag in the **Bob.Flags** field.

> *Changes Are Not Immediately Seen.* Neither these nor other changes are evident until **SortGList()** and then **DrawGList()** are called.

COMPLETE BOB EXAMPLE

This example must be linked with animtools.c and includes the header files animtools.h and animtools_proto.h. These files are listed at the end of the chapter.

```
/* bob.c
**
** SAS/C V5.10a
** lc -b1 -cfist -v -y bob.c
** blink FROM LIB:c.o bob.o animtools.o LIB LIB:lc.lib LIB:amiga.lib TO bob
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuitionbase.h>
#include <graphics/gfx.h>
```

```
#include <graphics/gfxbase.h>
#include <graphics/gels.h>
#include <libraries/dos.h>
#include <stdlib.h>
#include "animtools.h"

VOID bobDrawGList(struct RastPort *rport, struct ViewPort *vport);
VOID process_window(struct Window *win, struct Bob *myBob);
VOID do_Bob(struct Window *win);

struct GfxBase       *GfxBase;       /* pointer to Graphics library */
struct IntuitionBase *IntuitionBase; /* pointer to Intuition library*/
int return_code;
#define GEL_SIZE  4                  /* number of lines in the bob  */

/* Bob data - two sets that are alternated between.  Note that this */
/* data is at the resolution of the screen.                         */

/* data is 2 planes by 2 words by GEL_SIZE lines                    */
WORD chip bob_data1[2 * 2 * GEL_SIZE] =
        {
        /* plane 1 */
        0xffff, 0x0003, 0xfff0, 0x0003, 0xfff0, 0x0003, 0xffff, 0x0003,
        /* plane 2 */
        0x3fff, 0xfffc, 0x3ff0, 0x0ffc, 0x3ff0, 0x0ffc, 0x3fff, 0xfffc
        };

/* data is 2 planes by 2 words by GEL_SIZE lines                     */
WORD chip bob_data2[2 * 2 * GEL_SIZE] =
        {
        /* plane 1 */
        0xc000, 0xffff, 0xc000, 0x0fff, 0xc000, 0x0fff, 0xc000, 0xffff,
        /* plane 2 */
        0x3fff, 0xfffc, 0x3ff0, 0x0ffc, 0x3ff0, 0x0ffc, 0x3fff, 0xfffc
        };

NEWBOB myNewBob =                    /* Data for the new bob structure defined in animtools.h */
        {                            /* Initial image, WORD width, line height                */
        bob_data2, 2,  GEL_SIZE,     /* Image depth, plane pick, plane on off, VSprite flags  */
        2, 3, 0, SAVEBACK | OVERLAY, /* dbuf (0=false), raster depth, x,y position, hit mask, */
        0, 2, 160, 100,  0,0,        /* me mask                                               */
        };

struct NewWindow myNewWindow =
        {                            /* information for the new window */
        80, 20, 400, 150, -1, -1, CLOSEWINDOW | INTUITICKS,
        ACTIVATE | WINDOWCLOSE | WINDOWDEPTH | RMBTRAP,
        NULL, NULL, "Bob", NULL, NULL, 0, 0, 0, 0, WBENCHSCREEN
        };

/* Draw the Bobs into the RastPort. */
VOID bobDrawGList(struct RastPort *rport, struct ViewPort *vport)
{
SortGList(rport);
DrawGList(rport, vport);
/* If the GelsList includes true VSprites, MrgCop() and LoadView() here */
WaitTOF() ;
}

/* Process window and dynamically change bob: Get messages. Go away on CLOSEWINDOW.
** Update and redisplay bob on INTUITICKS. Wait for more messages.
*/
VOID process_window(struct Window *win, struct Bob *myBob)
{
struct IntuiMessage *msg;

FOREVER {
        Wait(1L << win->UserPort->mp_SigBit);
        while (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort)))
                {
                /* only CLOSEWINDOW and INTUITICKS are active */
                if (msg->Class == CLOSEWINDOW)
                        {
                        ReplyMsg((struct Message *)msg);
                        return;
                        }
```

```
                /* Must be INTUITICKS:  change x and y values on the fly.  Note:
                ** do not have to add window offset, Bob is relative to the
                ** window (sprite relative to screen).
                */
                myBob->BobVSprite->X = msg->MouseX + 20;
                myBob->BobVSprite->Y = msg->MouseY + 1;
                ReplyMsg((struct Message *)msg);
                }
        /* after getting a message, change image data on the fly */
        myBob->BobVSprite->ImageData =
                (myBob->BobVSprite->ImageData == bob_data1) ? bob_data2 : bob_data1;
        InitMasks(myBob->BobVSprite); /* set up masks for new image */
        bobDrawGList(win->RPort, ViewPortAddress(win));
        }
}


/* Working with the Bob: setup the GEL system, and get a new Bob (makeBob()).
** Add the bob to the system and display. Use the Bob.  When done, remove
** the Bob and update the display without the bob. Cleanup everything.
*/
VOID do_Bob(struct Window *win)
{
struct Bob          *myBob;
struct GelsInfo     *my_ginfo;

if (NULL == (my_ginfo = setupGelSys(win->RPort, 0x03)))
        return_code = RETURN_WARN;
else
        {
        if (NULL == (myBob = makeBob(&myNewBob)))
                return_code = RETURN_WARN;
        else
                {
                AddBob(myBob, win->RPort);
                bobDrawGList(win->RPort, ViewPortAddress(win));
                process_window(win, myBob);
                RemBob(myBob);
                bobDrawGList(win->RPort, ViewPortAddress(win));
                freeBob(myBob, myNewBob.nb_RasDepth);
                }
        cleanupGelSys(my_ginfo,win->RPort);
        }
}


/* Example bob program: First open up the libraries and a window. */
VOID main(int argc, char **argv)
{
struct Window       *win;

return_code = RETURN_OK;

if (NULL == (GfxBase = (struct GfxBase *)OpenLibrary(GRAPHICSNAME,37L)))
        return_code = RETURN_FAIL;
else
        {
        if (NULL == (IntuitionBase = (struct IntuitionBase *)OpenLibrary(INTUITIONNAME,37L)))
                return_code = RETURN_FAIL;
        else
                {
                if (NULL == (win = OpenWindow(&myNewWindow)))
                        return_code = RETURN_FAIL;
                else
                        {
                        do_Bob(win);
                        CloseWindow(win);
                        }
                CloseLibrary((struct Library *)IntuitionBase);
                }
        CloseLibrary((struct Library *)GfxBase);
        }
exit(return_code);
}
```

DOUBLE-BUFFERING

Double-buffering is the technique of supplying two different memory areas in which the drawing routines may create images. The system displays one memory space while drawing into the other area. This eliminates the ''flickering'' that is visible when a single display is being rendered into at the same time that it is being displayed.

> *Double-buffering For One Means Double-buffering For All.* If any of the Bobs is double-buffered, then *all* of them must be double-buffered.

To find whether a Bob is to be double-buffered, the system examines the pointer named **DBuffer** in the **Bob** structure. If this pointer has a value of NULL, the system does not use double-buffering for this Bob. For example:

```
myBob.DBuffer = NULL;    /* do this if this Bob is NOT double-buffered */
```

DBufPacket and Double-Buffering

For double-buffering, a place must be provided for the system to store the extra information it needs. The system maintains these data, and does not expect the application to change them. The **DBufPacket** structure consists of the following members:

BufY, BufX Lets the system keep track of where the object was located n the last frame'' (as compared to the **Bob** structure members called **oldY** and **oldX** that tell where the object was two frames ago). **BufY** and **BufX** provide for correct restoration of the background within the currently active drawing buffer.

BufPath Assures that the system restores the backgrounds in the correct sequence; it relates to the **VSprite** members **DrawPath** and **ClearPath**.

BufBuffer This field must be set to point to a buffer the same size as the Bob's **SaveBuffer**. This buffer is used to store the background for later restoration when the system moves the object. This buffer must be allocated from Chip memory.

To create a double-buffered Bob, execute a code sequence similar to the following:

```
struct Bob        myBob = {0};
struct DBufPacket myDBufPacket = {0};

/* Allocate a DBufPacket for myBob same size as previous example */
if (NULL != (myDBufPacket.BufBuffer = AllocRaster(48, 20 * 5)))
    {
    /* tell Bob about its double buff status */
    myBob.DBuffer = myDBufPacket;
    }
```

The example routines **makeBob()** and **freeBob()** in the animtools.c listing at the end of this chapter show how to correctly allocate and free a double-buffered Bob.

Collisions and GEL Structure Extensions

This section covers two topics that are applicable to all GELs: how to extend GEL data structures for your own purposes and how to detect collisions between GELs and other graphics objects.

DETECTING GEL COLLISIONS

All GELs, including VSprites, can participate in the software collision detection features of the graphics library. Simple Sprites must use *hardware* collision detection. See the *Amiga Hardware Reference Manual* for information about hardware collision detection.

Two kinds of collisions are handled by the system routines: GEL-to-boundary hits and GEL-to-GEL hits. You can set up as many as 16 different routines to handle different collision combinations; one routine to handle the boundary hits, and up to fifteen more to handle different inter-GEL hits.

You supply the actual collision handling routines, and provide their addresses to the system so that it can call them as needed (when the hits are detected). These addresses are kept in a collision handler table pointed to by the **CollHandler** field of the **GelsInfo** list. Which routine is called depends on the 16-bit **MeMask** and **HitMask** members of the **VSprite** structures involved in the collision.

When you call **DoCollision()**, the system goes through the **GelsInfo** list which, is constantly kept sorted by x, y position. If a GEL intersects the display boundaries and the GELs **HitMask** indicates it is appropriate, the boundary collision routine is called. When **DoCollision()** finds that two GELs overlap, it compares the **MeMask** of one with the **HitMask** of the other. If corresponding bits are set in both, it calls the appropriate inter-GEL collision routine at the table position corresponding to the bits in the **HitMask** and **MeMask**, as outlined below.

Preparing for Collision Detection

Before you can use the system to detect collisions between GELS, you must allocate and initialize a table of collision-detection routines and place the address of the table in the **GelsInfo.CollHandler** field. This table is an array of pointers to the actual routines that you have provided for your collision types. You must also prepare some members of the **VSprite** structure: **CollMask**, **BorderLine**, **HitMask**, and **MeMask**.

Building a Table of Collision Routines

The collision handler table is a structure, **CollTable**, defined in *<graphics/gels.h>*. It is accessed as the **CollHandler** member of the **GelsInfo** structure. The table only needs to be as large as the number of bits for which you wish to provide collision processing. It is safest, though, to allocate space for all 16 entries, considering the small amount of space required.

Call the routine **SetCollision()** to initialize the table entries that correspond to the **HitMask** and **MeMask** bits that you plan to use. Do not set any of the table entries directly, instead give the address to **SetCollision()** routine and let it handle the set up of the **GelsInfo.CollTable** field.

For example, **SetCollision()** could be called as follows:

```
ULONG           num;
VOID           (*routine)();
struct GelsInfo *GInfo;

VOID myCollisionRoutine(GELA, GELB)   /* sample collision routine */
struct VSprite *GELA;
struct VSprite *GELB;
{
    /* process GELs here - GELA and GELB point to the base VSprites of */
    /* the GELs, you can use the user extensions to identify what hit  */
    /* (if you need the info).                                         */
}

/* GelsInfo must be allocated and initialized */

routine = myCollisionRoutine;

SetCollision(num, routine, GInfo)
```

The **num** argument is the collision table vector number (0-15). The **(*routine)()** argument is a pointer to your collision routine. And the **GInfo** argument is a pointer to the **GelsInfo** structure.

VSprite Collision Mask

The **CollMask** member of the **VSprite** is a pointer to a memory area allocated for holding the collision mask of that GEL. This area must be in Chip memory and its size is the equivalent of one bitplane of the GEL's image. The collision mask is *usually* the same as the shadow mask of the GEL, formed from a logical-OR combination of all planes of the image. The following figure shows an example collision mask.

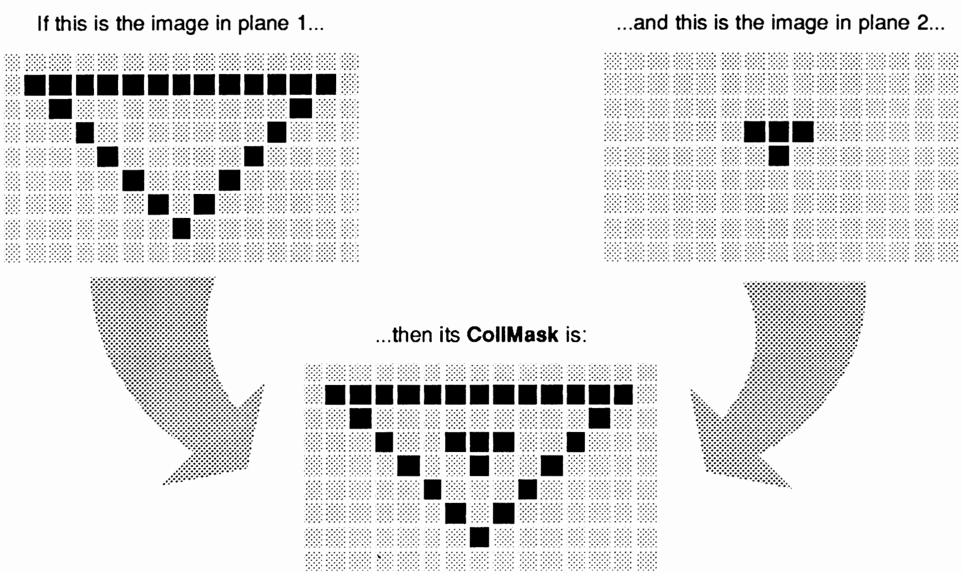


Figure 28-3: A Collision Mask

Alternatively, you may have a collision mask that is not derived from the image. In this case, the actual image isn't relevant. The system will not register collisions unless the other objects touch the collision mask. If the collision mask is smaller than the image, other objects will pass through the edges without a collision.

VSprite BorderLine

For faster collision detection, the system uses the **BorderLine** member of the **VSprite** structure. **BorderLine** specifies the location of the horizontal logical-OR combination of all of the bits of the object. It may be compared to taking the whole object's shadow/collision mask and squishing it down into a single horizontal line. You provide the system with a place to store this line. The size of the data area you allocate must be at least as large as the image width.

In other words, if it takes three 16-bit words to hold one line of a GEL, then you must reserve three words for the **BorderLine**. In the VSprite examples, the routine **makeVSprite()** correctly allocates and initializes the collision mask and borderline. For example:

```
WORD myBorderLineData[3];    /* reserve space for BorderLine for this Bob */

myVSprite.BorderLine = myBorderLineData;   /* tell the system where it is */
```

Here is a sample of an object and its **BorderLine** image:

```
011000001100  Object
001100011000
001100011000
000110110000
000010100000

011110111100  BorderLine image
```

Using this squished image, the system can quickly determine if the image is touching the left or rightmost boundary of the drawing area.

To establish default **BorderLine** and **CollMask** data, call the **InitMasks()** function.

VSprite HitMask and MeMask

Software collision detection is independently enabled and disabled for each GEL. Further, you can specify which of 16 possible collision routines you wish to have automatically executed. **DoCollision()**, in addition to sensing an overlap between objects, uses these masks to determine which routine (if any) the system will call when a collision occurs.

When the system determines a collision, it performs a logical-AND of the **HitMask** of the upper-leftmost object in the colliding pair with the **MeMask** of the lower-rightmost object of the pair. The bits that are 1s after the logical-AND operation choose which one of the 16 possible collision routines to perform.

- If the collision is with the boundary, bit 0 is always a 1 and the system calls the collision handling routine number 0. Always assign the routine that handles boundary collisions to vector 0 in the collision handling table. The system uses the flag called BORDERHIT to indicate that an object has landed on or moved beyond the outermost bounds of the drawing area (the edge of the clipping region). The VSprite example earlier in this chapter uses collision detection to check for border hits.
- If any one of the other bits (1 to 15) is set, then the system calls your collision handling routine corresponding to the bit set.
- If more than one bit is set in both masks, the system calls the vector corresponding to the rightmost (the least significant) bit *only*.

Using HitMask and MeMask

This section illustrates the use of the **HitMask** and **MeMask** to define one type of collision.

Suppose there are two classes of objects that you wish to control on the display: ENEMYTANK and MYMISSILE. Objects of class ENEMYTANK should be able to pass across one another without registering any collisions. Objects of class MYMISSILE should also be able to pass across one another without collisions. However, when MYMISSILE and ENEMYTANK collide, the system should generate a call to a collision routine.

Choose a pair of collision detect bits not yet assigned within **MeMask**, one to represent ENEMYTANK, the other to represent MYMISSILE. You will use the same two bits in the corresponding **HitMask**. In this example, bit 1 represents ENEMYTANK objects and bit 2 represents MYMISSLE objects.

	MeMask	HitMask
Bit Number	**2 1**	**2 1**
ENEMYTANK 1	0 1	1 0
ENEMYTANK 2	0 1	1 0
MYMISSILE	1 0	0 1

In the **MeMask**, bit 1 is set to indicate that this object is an ENEMYTANK. Bit 2 is clear (zero) indicating this object is *not* a MYMISSILE object. In the **HitMask** for ENEMYTANK objects, bit 1 is clear (zero) which means, "I will not register collisions with other ENEMYTANK objects." However, bit 2 is set (one) which means, "I *will* register collisions with MYMISSILE objects."

Thus when a call to **DoCollision()** occurs, for any objects that appear to be colliding, the system ANDs the **MeMask** of one object with the **HitMask** of the other object. If there are non-zero bits present, the system will call one of your collision routines.

In this example, suppose that the system senses a collision between ENEMYTANK 1 and ENEMYTANK 2. Suppose also that ENEMYTANK 1 is the top/leftmost object of the pair. Here is the way that the collision testing routine performs the test to see if the system will call any collision-handling routines:

Bit Number	**2**	**1**
ENEMYTANK 1 **MeMask**	0	1
ENEMYTANK 2 **HitMask**	1	0
Result of logical-AND	0	0

Therefore, the system does not call a collision routine. But suppose that **DoCollision()** finds an overlap between ENEMYTANK 1 and MYMISSILE, where MYMISSILE is the top/leftmost of the pair:

Bit Number	**2**	**1**
MYMISSILE **MeMask**	1	0
ENEMYTANK 2 **HitMask**	1	0
Result of logical-AND	1	0

Therefore, the system calls the collision routine at position 2 in the table of collision-handling routines.

SETTING UP FOR BOUNDARY COLLISIONS

To specify the region in the playfield that the system will use to define the outermost limits of the GEL boundaries, you use these **GelsInfo** members: **topmost**, **bottommost**, **leftmost**, and **rightmost**. The **DoCollision()** routine tests these boundaries when determining boundary collisions within this **RastPort**. They have nothing whatsoever to do with graphical clipping. Graphical clipping makes use of the **RastPort**'s clipping rectangle.

Here is a typical program segment that assigns the members correctly (for boundaries 50, 100, 80, 240). It assumes that you already have a **RastPort** structure pointer named **myRastPort**.

```
myRastPort->GelsInfo->topmost    = 50;
myRastPort->GelsInfo->bottommost = 100;
myRastPort->GelsInfo->leftmost   = 80;
myRastPort->GelsInfo->rightmost  = 240;
```

Parameters To Your Boundary Collision Routine

During the operation of the **DoCollision()** routine, if you have enabled boundary collisions for a GEL (by setting the least significant bit of its **HitMask**) and it has crossed a boundary, the system calls the boundary routine you have defined. The system will call the routine once for every GEL that has hit, or gone outside of the boundary. The system will call your routine with the following two arguments:

- A pointer to the **VSprite** structure of the GEL that hit the boundary
- A flag word containing one to four bits set, representing top, bottom, left and right boundaries, telling you which of the boundaries it has hit or exceeded. To test these bits, compare to the constants TOPHIT, BOTTOMHIT, LEFTHIT, and RIGHTHIT.

See the VSprite example given earlier for an example of using boundary collision.

Parameters To Your Inter-GEL Collision Routines

If, instead of a GEL-to-boundary collision, **DoCollision()** senses a GEL-to-GEL collision, the system calls your collision routine with the following two arguments:

- Address of the VSprite that is the uppermost (or leftmost if y coordinates are identical) GEL of a colliding pair.
- Address of the VSprite that is the lowermost (or rightmost if y coordinates are identical) GEL of the pair.

Handling Multiple Collisions

When multiple elements collide within the same display field, the following set of sequential calls to the collision routines occurs:

- The system issues each call in a sorted order for GELs starting at the upper left-hand corner of the screen and proceeding to the right and down the screen.
- For any colliding GEL pair, the system issues only one call, to the collision routine for the object that is the topmost and leftmost of the pair.

ADDING USER EXTENSIONS TO GEL DATA STRUCTURES

This section describes how to expand the size and scope of the **VSprite**, **Bob** and **AnimOb** data structures. In the definition for these structures, there is an item called **UserExt** at the end of each. If you want to expand these structures (to hold your own special data), you define the **UserExt** member before the *<graphics/gels.h>* file is included. If this member has already been defined when the *<graphics/gels.h>* file is parsed, the compiler preprocessor will extend the structure definition automatically. If these members have not been defined, the system will make them SHORTs, and you may still consider these as being reserved for your private use.

To show the kind of use you can make of this feature, the example below adds speed and acceleration figures to each GEL by extending the **VSprite** structure. When your collision routine is called, it could use these values to transfer energy between the two colliding objects (say, billiard balls). You would have to set up additional routines, executed between calls to **DoCollision**(), that would add the values to the GELs position appropriately. You could do this with code similar o the following:

```
struct myInfo
    {
    short xvelocity;
    short yvelocity;
    short xaccel;
    short yaccel;
    };
```

These members are for example only. You may use any definition for your user extensions. You would then also provide the following line, to extend the **VSprites** structure, use:

```
/* Redefine VUserStuff for my own use. */
#define VUserStuff struct myInfo
```

To extend the **Bobs** structure, use:

```
#define BUserStuff struct myInfo
```

To extend the **AnimObs** structure, use:

```
#define AUserStuff struct myInfo
```

When the system is compiling the *<graphics/gels.h>* file with your program, the compiler preprocessor substitutes "struct myInfo" everywhere that **UserExt** is used in the header. The structure is thereby customized to include the items you wish to associate with it.

> *Typedef Cannot Be Used.* You cannot use the C-language construct **typedef** for the above statements. If you want to substitute your own data type for one of the **UserStuff** variables, you must use a #define.

Animation with GELs

An animation sequence is composed of a series of drawings. Each drawing differs from the preceding one so that when they are arranged in a stack and viewed sequentially, the images appear to flow naturally.

In classic film animation, image drawing is done in two stages. The background for each scene is painted just once. Then, the cartoon characters and any other foreground objects are painted on transparent sheets of celluloid called *cells* which are placed over the background. With cells, animation can be achieved by redrawing only the parts of the scene that move while the background stays the same. Animation on the Amiga works similarly. The background is formed by the playfield while the objects that move can be conveniently handled with the GELs system.

ANIMATION DATA STRUCTURES

There are two main data structures involved in Amiga animation: **AnimComp** and **AnimOb**.

The **AnimComp** (Animation Component), is an extension of the **Bob** structure discussed in the previous section. An **AnimComp** provides a convenient way to link together a series of images so that they can be sequenced automatically, and so multiple sequences can be grouped together. An **AnimComp** is analogous to one sheet of celluloid representing a single image to be placed over the background.

```
struct AnimComp
{
    WORD Flags;                 /* AnimComp flags for system & user */
    WORD Timer;
    WORD TimeSet;
    struct AnimComp  *NextComp;
    struct AnimComp  *PrevComp;
    struct AnimComp  *NextSeq;
    struct AnimComp  *PrevSeq;
    WORD (*AnimCRoutine)();
    WORD YTrans;
    WORD XTrans;
    struct AnimOb    *HeadOb;   /* Pointer back to the controlling AnimOb     */
    struct Bob        *AnimBob;  /* Underlying Bob structure for this AnimComp */
};
```

The **AnimComp** structure contains pointers, **PrevSeq** and **NextSeq**, that lets you group these cells into stacks that will be viewed sequentially. The **AnimComp** structure also has **PrevComp** and **NextComp** pointers that let you group stacks into complex objects containing multiple independently moving parts.

The second animation data structure is the **AnimOb** which provides the variables needed for overall control over a group of **AnimComps**. The **AnimOb** itself contains no imagery; it simply provides a common reference point for the sequenced images and specifies how the system should move that point.

```
struct AnimOb
{
    struct AnimOb   *NextOb, *PrevOb;
    LONG Clock;
    WORD AnOldY, AnOldX;          /* old y,x coordinates         */
    WORD AnY, AnX;                /* y,x coordinates of the AnimOb*/
    WORD YVel, XVel;              /* velocities of this object    */
    WORD YAccel, XAccel;          /* accelerations of this object */
    WORD RingYTrans, RingXTrans;    /* ring translation values      */
    WORD (*AnimORoutine)();       /* address of user procedure    */
    struct AnimComp *HeadComp;      /* pointer to first component   */
    AUserStuff AUserExt;          /* AnimOb user extension        */
};
```

These structures can be used in various ways. A simple animation of a rotating ball could be created with three or four **AnimComps** linked together in a circular list by their **NextSeq** and **PrevSeq** fields. The system displays the initial **AnimComp** (the "top of the stack"), then switches to the **AnimComp** pointed to by **NextSeq**, and then switches to *its* **NextSeq** and so on until it reaches the end of the sequence. The sequence starts over again automatically if the last **AnimComp.NextSeq** points back to the first **AnimComp** in the stack.

For a more complex animation of a walking human, you could use five stacks, i.e., five circular lists of **AnimComps**; four stacks for the arms and legs and a single stack for the head and torso. To group these stacks into one cohesive unit showing a human figure, you use the **PrevComp** and **NextComp** pointers in the **AnimComp** structure. All the stacks would also share a common **AnimOb**, so that the combined sequences can be moved as a single object.

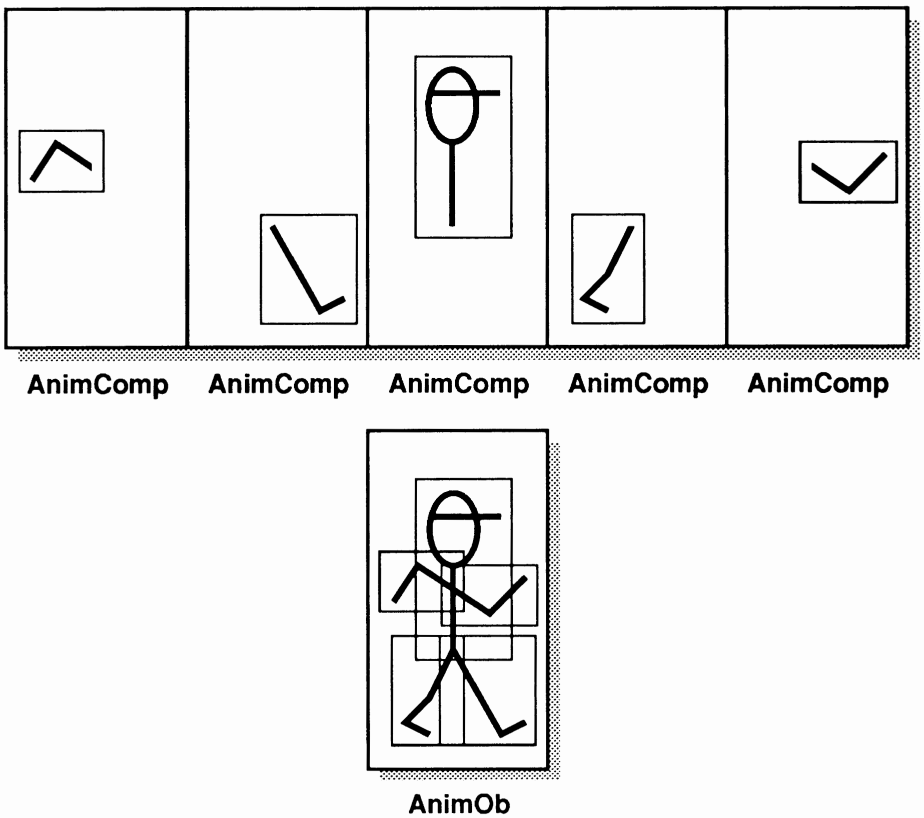


Figure 28-4: Linking AnimComps For a Multiple Component AnimOb

ANIMATION TYPES

The GELs system provides several ways of setting up automatic animation, loosely based on some categories of movement in real life. Some things (like balls or arrows) can move independently of the background, and look even more realistic if they tumble or rotate as they move; other things (like worms, wheels, and people) must be anchored to the background, or they will appear to *slide* unnaturally.

The system software allows these types of animation through simple motion control, motion control with sequenced drawing, and sequenced drawing using Ring Motion Control.

Simple Motion Control

To produce motion of a simple object, such as a ball, the object is simply moved relative to a background display, a little at a time. This is simple motion control, and can be accomplished with one **AnimComp** and one **AnimOb**, by simply changing the **AnimOb**'s position every N video frames. The apparent speed of the object is a combination of how often it is moved (every frame, every other frame, etc.) and how far it is moved (how much the **AnimOb**'s **AnX** and **AnY** are changed).

Sequenced Drawing

To make the ball appear to rotate is a little more complex. To produce apparent movement within the image, sequencing is used. This is done by having a stack of **AnimComp**'s that are laid down one after the other, a frame at a time. The stack can be arranged in a circular list for repeating movement. So, when you combine a sequence of drawings using **AnimComps** with simple motion control using an **AnimOb**, you can perform more complex animations such as having a rotating ball bounce around.

Ring Motion Control

Making a worm appear to crawl is similar to the rotating ball. There is still a stack of **AnimComps** that are sequenced automatically, and one controlling **AnimOb**. But each **AnimCop** image is drawn so that it appears to move relative to an internal point that remains stationary throughout the stack. So instead of the **AnimOb**'s common reference point moving in each frame, you tell the system how far to move only *at the end* of each **AnimComp** sequence.

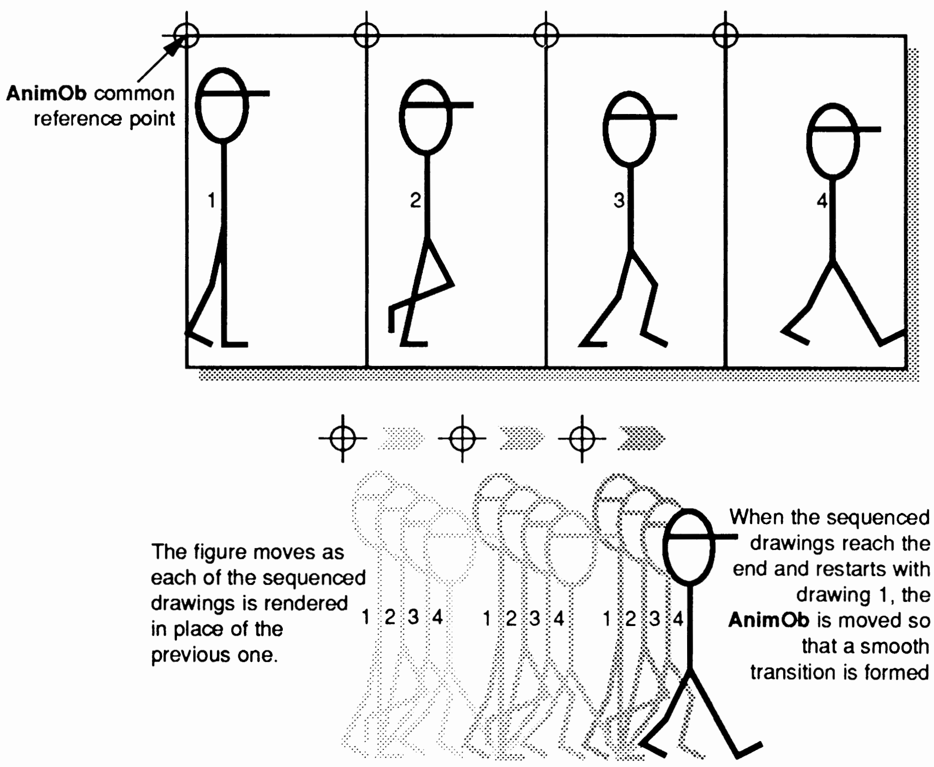


Figure 28-5: Ring Motion Control

As illustrated in the figure at left, the sequence of events for Ring Motion Control look like this:

Draw AnimComp1, Draw AnimComp2, Draw AnimComp3, Move AnimOb,
Draw AnimComp1, Draw AnimComp2, Draw AnimComp3, Move AnimOb,
Draw AnimComp1...

SPECIFYING ANIMATION COMPONENTS

For each **AnimComp**, you initially specify:

- A pointer to the **AnimComp**'s controlling **AnimOb**.
- Initial and alternate views, their timing and order.
- The initial inter-component drawing priorities (for multiple **AnimComp** sequences, this specifies which sequence to display frontmost).
- A pointer to a special animation routine related to this component (optional).
- Your own extensions to this structure (optional).

Sequencing AnimComps

To specify the sequencing of **AnimComp** images, the pointers called **PrevSeq** and **NextSeq** are used to build a doubly-linked list. The sequence can be made circular (and usually is) by linking the first and last **AnimComps** in the sequence: the **NextSeq** of the last **AnimComp** must point back to the first **AnimComp**, and the **PrevSeq** of the first **AnimComp** must point to the last **AnimComp**. If the list is a loop, then the system will continue to cycle through the list until it is stopped. If the list is not a loop, then the program must act to restart the sequence after the last item is displayed. The **AnimCRoutine** field of the last **AnimComp** can be used to do this.

Position of an AnimComp

To specify the placement of each **AnimComp** relative to its controlling **AnimOb**, you set the **AnimComp** members **XTrans** and **YTrans**. These values can be positive or negative.

The system is designed so that only one of the **AnimComps** in any given sequence is "active" (being displayed) at a given point in time. It is the only image in the sequence that is (or is about to be) linked into the **GelsInfo** list. The Timer determines how long each Component in the sequence remains active, as described below.

Specifying Time for Each Image

The **AnimComp** members **Timer** and **TimeSet** are used to specify how long the system should keep each sequential image on the screen.

When the system makes an animation component active, it copies the value you have put in the **TimeSet** member into the **Timer** member. As the animation proceeds, the system decrements **Timer**; as long as it is greater than zero, then that **AnimComp** remains active. When the **Timer** value reaches zero, the system makes the next **AnimComp** in the sequence active, and the process repeats.

If you initialize the value in **TimeSet** to zero, the system will *not sequence this component at all* (and **Timer** will remain zero).

Linking Multiple AnimComp Sequences

When an **AnimOb** is built from multiple **AnimComp** sequences, the sequences are linked together by the the **PrevComp** and **NextComp** fields of the **AnimComps**. These pointers must be initialized *only in the initial* **AnimComp** *of each sequence*. The other components that are not initially active should have their **PrevComp** and **NextComp** pointers set to NULL.

> *Do Not Use Empty Fields.* You cannot store data in the empty **PrevComp** and **NextComp** fields. As the system cycles through the **AnimComps**, the **NextComp** and **PrevComp** fields are set to NULL when an old **AnimComps** is replaced by a new **AnimComp**. The new **AnimComp** is then linked in to the list of sequences *in place of the old one.*

Component Ordering

The **PrevSeq**, **NextSeq**, **PrevComp** and **NextComp** linkages have no bearing on the order in which **AnimComps** in any given video frame are drawn. To specify the inter-component priorities (so that the closest objects appear frontmost) the **Before** and **After** pointers in the initially active **AnimComp**'s underlying **Bob** structure are linked in to the rest of the system, as described previously in the discussion of Bobs.

This setup needs to be done once, *for the initially active* **AnimComp***s of the* **AnimOb** *only.*

The animation system adjusts the **Before** and **After** pointers of all the underlying **Bob** structures to constantly maintain the inter-component drawing sequence, even though different components are being made active as sequencing occurs.

These pointers also assure that one complete *object* always has priority over another object. The Bob Before and After pointers are used to link together the last **AnimComp**'s Bob of one AnimOb to the first AnimComp's Bob of the next AnimOb.

SPECIFYING THE ANIMATION OBJECT

For each **AnimOb**, you initially specify:

- The starting position of this object
- Its velocity and acceleration (optional).
- A pointer to the first of its **AnimComps**.
- A pointer to a special animation routine related to this object (optional).
- Your own extensions to this structure (optional).

Linking the AnimComp Sequences to the AnimOb

Within each **AnimOb** there may be one or more **AnimComp** sequences. The **HeadComp** of the **AnimOb** points to the first **AnimComp** in the list of sequences.

Each sequence is identified by its ''active'' **AnimComp**. There can only be one active **AnimComp** in each sequence. The sequences are linked together by their active **AnimComps**; for each of these the **NextComp** and **PrevComp** fields link the sequences together to create a list. The first sequence in the list (**HeadComp** of the **AnimOb**), has its **PrevComp** set to NULL. The last sequence in the list has its **NextComp** set to NULL. None of the inactive **AnimComps** should have **NextComp** or **PrevComp** fields set.

To find the active **AnimComp** at run time, you can look in the **AnimOb**'s **HeadComp** field. To find the active **AnimComp** from any another **AnimComp**, use the **HeadOb** field to find the controlling **AnimOb** first and then look in its **HeadComp** field to find the active **AnimComp**.

The figure below shows all the linkages in data structures needed to create the animation GELs.

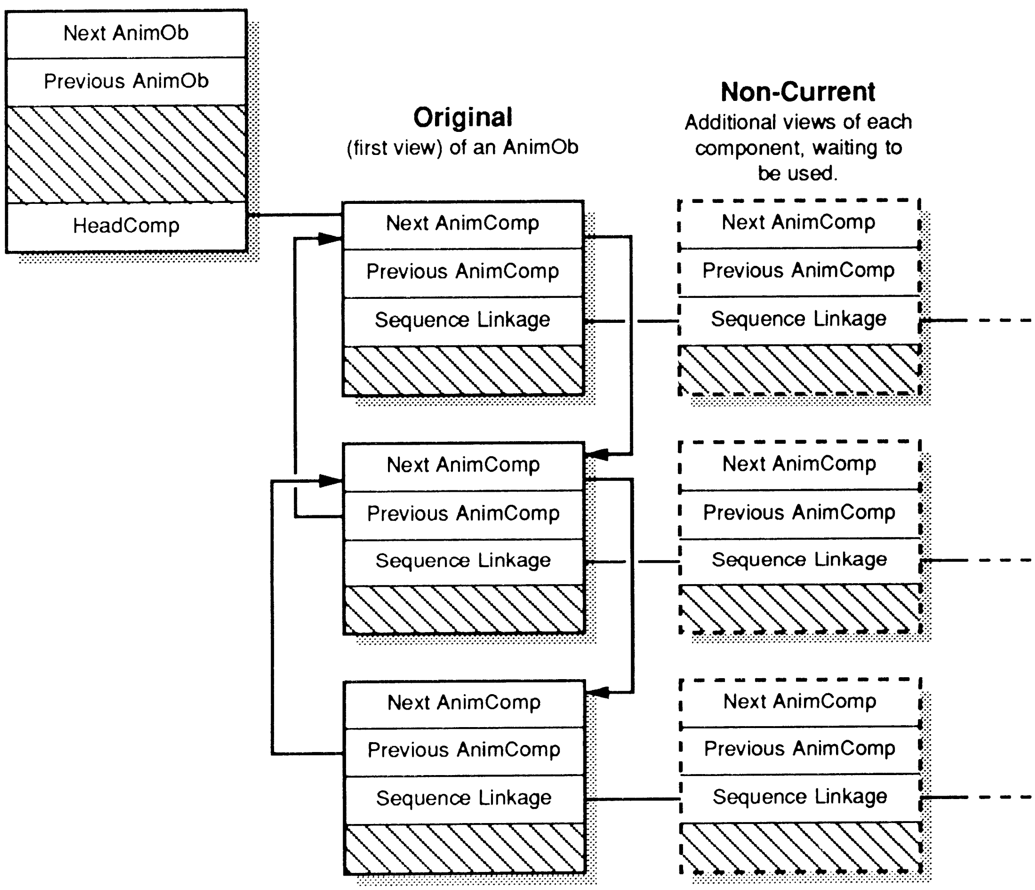


Figure 28-6: Linking of an AnimOb

Position of an AnimOb

To position the object and its component parts, use the **AnimOb** structure members **AnX** and **AnY**. The following figure illustrates how each component has its own offset from the **AnimOb**'s common reference point.

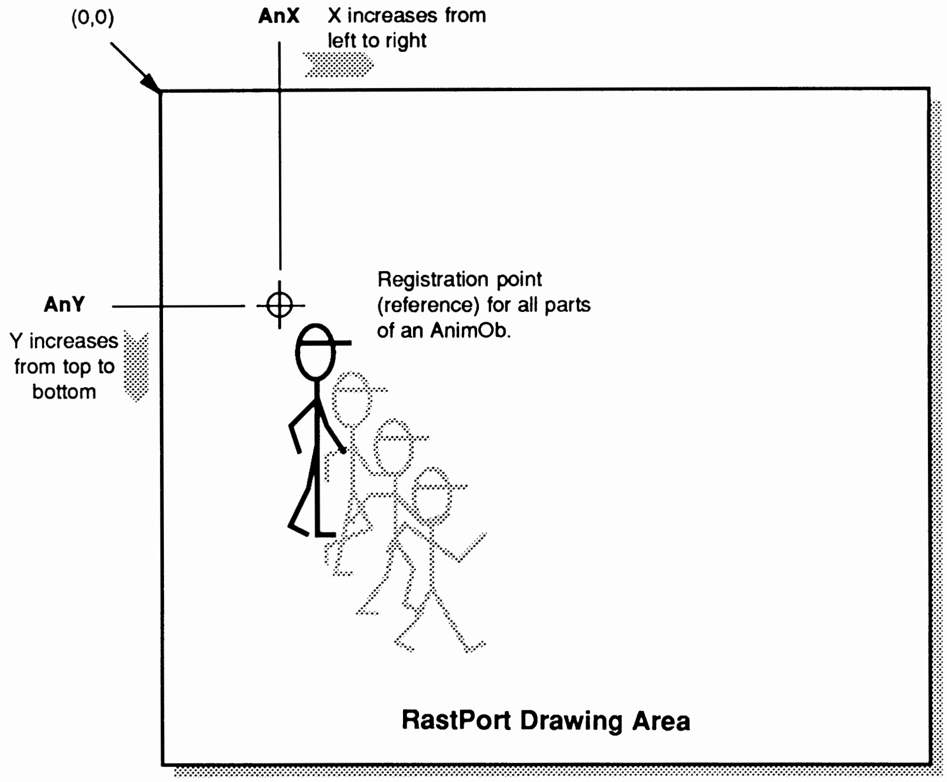


Figure 28-7: Specifying an AnimOb Position

When you change the animation object's **AnX** and **AnY**, all of the component parts will be redrawn relative to it the next time **DrawGList()** is called.

Setting Up Simple Motion Control

In this form of animation, you can specify objects that have independently controllable velocities and accelerations in the X and Y directions. Components can still sequence.

The variables that control this motion are located in the **AnimOb** structure and are called:

- **YVel, XVel**—the velocities in the y and x directions. These values are added to the position values on each call to **Animate()** (see below).

- **YAccel, XAccel**—the accelerations in the xa nd y directions. These values are added to the velocity values on each call to **Animate()** (see below). The velocity values are updated *before* the position values.

Setting Up Ring Motion Control

To make a given component *trigger* a move of the **AnimOb** you set the RINGTRIGGER bit of that **AnimComp's Flags** field. When the system software encounters this flag, it adds the values of **RingXTrans** and **RingYTrans** to the **AnX** and **AnY** values of the controlling **AnimOb**. The *next* time you execute **DrawGList()**, the drawing sequence will use the new position.

You usually set RINGTRIGGER in only one of the animation components in a sequence (the last one); however, you can use this flag and the translation variables any way you wish.

Using Sequenced Drawing and Motion Control

If you are using Ring Motion Control, you will probably set the velocity and acceleration variables to zero. For instance, consider the example of a person walking. With Ring Motion Control, as each foot falls it is positioned on the ground exactly where originally drawn. If you included a velocity value, the person's foot would not be stationary with respect to the ground, and the person would appear to ''skate'' rather than walk. If you set the velocity and acceleration variables at zero, you avoid this problem.

When the system activates a new **AnimComp**, it checks the **Flags** field to see if the RINGTRIGGER bit is set. If so, the system adds **RingYTrans** and **RingXTrans** to **AnY** and **AnX** respectively.

THE ANIMKEY

The system uses one pointer, known as the AnimKey, to keep track of all the **AnimObs** via their **PrevOb** and **NextOb** linkage fields. The AnimKey acts as the anchor for the list of **AnimObs** you are using and is initialized with code such as the following:

```
struct AnimOb *animKey;

InitAnimate(&animKey);  /* Only do this once to initialize the AnimOb list */
```

As each new object is added (via **AddAnimOb()**), it is linked in at the beginning of the list, so AnimKey will always point to the object most recently added to the list. To search forward through the list, start with the AnimKey and move forward on the **NextOb** link. Continue to move forward until the **NextOb** is NULL, indicating the end of the list. The **PrevOb** link will allow you to move back to a previous object.

> *Set Up PrevOb and NextOb Correctly.* It is important that the **NextOb** link of the last object is NULL, and that the **PrevOb** of the first object is NULL. In fact, the system expects the animation object lists to be *exactly* the way that they are described above. If they are not, the system will produce unexpected results.

ADDING ANIMATION OBJECTS

Use the routine **AddAnimOb()** to add animation objects to the controlled object list. This routine will link the **PrevOb** and **NextOb** pointers to chain all the **AnimObs** that the system is controlling.

```
struct RastPort myRPort;
struct AnimOb myAnimOb;
struct AnimOb *animKey;  /* Must be initialized with InitAnimate() */

AddAnimOb(&myAnimOb, &animKey, &myRPort);
```

MOVING THE OBJECTS

When you have defined all of the structures and have established all of the links, you can call the **Animate()** routine to move the objects. **Animate()** adjusts the positions of the objects as described above, and calls the various subroutines (**AnimCRoutines** and **AnimORoutines**) that you have specified.

After the system has completed the **Animate()** routine, some GELs may have been moved, so the **GelsInfo** list order may possibly be incorrect. Therefore, the list must be re-sorted with **SortGList()** before passing it to a system routine.

If you are using collision detection, you then perform **DoCollision()**. Your collision routines may also have an effect on the relative position of the GELs. Therefore, you should again call **SortGList()** to assure that the system correctly orders the objects before you call **DrawGList()**. When you call **DrawGList()**, the system renders all the GELs it finds in the **GelsInfo** list and any changes caused by the previous call to **Animate()** can then be seen.

This is illustrated in the following typical call sequence:

```
struct AnimOb **myAnimKey;
struct RastPort *rp;
struct ViewPort *vp;

/* ... setup of graphics elements and objects */

Animate(myAnimKey, rp);       /* "move" objects per instructions */
SortGList(rp);                /*  put them in order */
DoCollision(rp);              /*  software collision detect/action */
SortGList(rp);                /*  put them back into right order */
DrawGList(vp, rp);            /*  draw into current RastPort */
```

YOUR OWN ANIMATION ROUTINE CALLS

The **AnimOb** and **AnimComp** structures can include pointers for your own routines that you want the system to call. These pointers are stored in the **AnimOb**'s **AnimORoutine** field and in the **AnimComp**'s **AnimCRoutine** field, respectively.

When **Animate()** is called, the system performs the following steps for every **AnimOb** in the AnimKey list:

- Updates the **AnimOb**'s location and velocities.
- Calls the **AnimOb.AnimORoutine** routine if one is supplied.
- The for each **AnimComp** of the **AnimOb**:
    - If this sequence times out, switches to the new **AnimComp**.
    - Calls the **AnimComp.AnimCRoutine** if one is supplied.
    - Sets the underlying **VSprite**'s x,y coordinates.

If you want a routine to be called, you put the address of the routine in either **AnimComp.AnimCRoutine** or **AnimOb.AnimORoutine** member as needed. If no routine is to be called, you must set these fields to NULL. Your routines will be passed one parameter, a pointer to the **AnimOb** or **AnimComp** it was related to. You can use the user structure extensions discussed earlier to hold the variables you need for your own routines.

For example, if you provide a routine such as this:

```
VOID MyOCode(struct AnimOb *anOb)
{
/* whatever needs to be done */
}
```

Then, if you put the address of the routine in an **AnimOb** structure:

```
myAnimOb.AnimORoutine = MyOCode;
```

MyOCode() will be called with the address of this **AnimOb** when **Animate()** processes this **AnimOb.**

STANDARD GEL RULES STILL APPLY

Before you use the animation system, you must have called the routine **InitGels()**. The section called "Bob Priorities" describes how the system maintains the list of GELs to draw on the screen according to their various data fields. The animation system selectively adds GELs to and removes GELs from this list of screen objects during the **Animate()** routine. On the next call to **DrawGList()**, the system will draw the GELs in the list into the selected **RastPort**.

ANIMATIONS SPECIAL NUMBERING SYSTEM

Velocities and accelerations can be either positive or negative. The system treats the velocity, acceleration and Ring values as fixed-point binary fractions, with the decimal point at position 6 in the word. That is: `vvvvvvvvvv.ffffff` where `v` stands for actual values that you add to the x or y (**AnX**, **AnY**) positions of the object for each call to **Animate()**, and `f` stands for the fractional part. By using a fractional part, you can specify the speed of an object in increments as precise as 1/64th of an interval.

If you set the value of **XVel** at 0x0001, it will take 64 calls to the **Animate()** routine before the system will modify the object's x coordinate position by a step of one. The system constant ANFRACSIZE can be used to shift values correctly. So if you set the value to (1 << ANFRACSIZE), it will be set to 0x0040, the value required to move the object one step per call to **Animate()**. The system constant ANIMHALF can be used if you want the object to move every other call to **Animate()**.

Each call you make to **Animate()** simply adds the value of **XAccel** to the current value of **XVel**, and **YAccel** to the current value of **YVel**, modifying these values accordingly.

ANIMTOOLS.H AND ANIMTOOLS.C

Here is the listing of the animtools.h header file and the animtools.c link file used by the examples in this chapter. The **makeSeq()** and **makeComp()** subroutines here demonstrate how to use the GELs animation system.

```
/* animtools.h */
#ifndef GELTOOLS_H
#define GELTOOLS_H

/*
** These data structures are used by the functions in animtools.c to
** allow for an easier interface to the animation system.
*/

/* Data structure to hold information for a new VSprite.            */
```

```
typedef struct newVSprite {
        WORD            *nvs_Image;      /* image data for the vsprite   */
        WORD            *nvs_ColorSet;   /* color array for the vsprite  */
        SHORT            nvs_WordWidth;  /* width in words               */
        SHORT            nvs_LineHeight; /* height in lines              */
        SHORT            nvs_ImageDepth; /* depth of the image           */
        SHORT            nvs_X;          /* initial x position           */
        SHORT            nvs_Y;          /* initial y position           */
        SHORT            nvs_Flags;      /* vsprite flags                */
        USHORT           nvs_HitMask;    /* Hit mask.                    */
        USHORT           nvs_MeMask;     /* Me mask.                     */
        } NEWVSPRITE;

/* Data structure to hold information for a new Bob.               */
typedef struct newBob {
        WORD        *nb_Image;       /* image data for the bob       */
        SHORT        nb_WordWidth;   /* width in words               */
        SHORT        nb_LineHeight;  /* height in lines              */
        SHORT        nb_ImageDepth;  /* depth of the image           */
        SHORT        nb_PlanePick;   /* planes that get image data   */
        SHORT        nb_PlaneOnOff;  /* unused planes to turn on     */
        SHORT        nb_BFlags;      /* bob flags                    */
        SHORT        nb_DBuf;        /* 1=double buf, 0=not          */
        SHORT        nb_RasDepth;    /* depth of the raster          */
        SHORT        nb_X;           /* initial x position           */
        SHORT        nb_Y;           /* initial y position           */
        USHORT       nb_HitMask;     /* Hit mask.                    */
        USHORT       nb_MeMask;      /* Me mask.                     */
        } NEWBOB ;

/* Data structure to hold information for a new animation component.      */
typedef struct newAnimComp {
        WORD   (*nac_Routine)(); /* routine called when Comp is displayed.   */
        SHORT    nac_Xt;         /* initial delta offset position.           */
        SHORT    nac_Yt;         /* initial delta offset position.           */
        SHORT    nac_Time;       /* Initial Timer value.                     */
        SHORT    nac_CFlags;     /* Flags for the Component.                 */
        } NEWANIMCOMP;

/* Data structure to hold information for a new animation sequence.        */
typedef struct newAnimSeq {
        struct AnimOb  *nas_HeadOb; /* common Head of Object.                */
        WORD   *nas_Images;         /* array of Comp image data              */
        SHORT  *nas_Xt;             /* arrays of initial offsets.            */
        SHORT  *nas_Yt;             /* arrays of initial offsets.            */
        SHORT  *nas_Times;          /* array of Initial Timer value.         */
        WORD (**nas_Routines)();    /* Array of fns called when comp drawn   */
        SHORT    nas_CFlags;        /* Flags for the Component.              */
        SHORT    nas_Count;         /* Num Comps in seq (= arrays size)      */
        SHORT    nas_SingleImage;   /* one (or count) images.                */
        } NEWANIMSEQ;

#define INTUITIONNAME "intuition.library" /* intuitionbase.h does not define a library name. */

#include "animtools_proto.h"            /* Include Prototyping. */
#endif


/* animtools_proto.h */
#include        <clib/dos_protos.h>
#include        <clib/exec_protos.h>
#include        <clib/graphics_protos.h>
#include        <clib/intuition_protos.h>

struct GelsInfo *setupGelSys(struct RastPort *rPort, BYTE reserved);
VOID            cleanupGelSys(struct GelsInfo *gInfo, struct RastPort *rPort);
struct VSprite  *makeVSprite(NEWVSPRITE *nVSprite);
struct Bob      *makeBob(NEWBOB *nBob);
struct AnimComp *makeComp(NEWBOB *nBob, NEWANIMCOMP *nAnimComp);
struct AnimComp *makeSeq(NEWBOB *nBob, NEWANIMSEQ *nAnimSeq);
VOID            freeVSprite(struct VSprite *vsprite);
VOID            freeBob(struct Bob *bob, LONG rasdepth);
VOID            freeComp(struct AnimComp *myComp, LONG rasdepth);
VOID            freeSeq(struct AnimComp *headComp, LONG rasdepth);
VOID            freeOb(struct AnimOb *headOb, LONG rasdepth);
```

```
/* animtools.c
**
** This file is a collection of tools which are used with the VSprite, Bob and Animation
** system software. It is intended as a useful EXAMPLE, and while it shows what must be
** done, it is not the only way to do it.  If Not Enough Memory, or error return, each
** cleans up after itself before returning.  NOTE that these routines assume a very specific
** structure to the GEL lists.  Make sure that you use the correct pairs together
** (i.e. makeOb()/freeOb(), etc.)
**
** Compile with SAS/C 5.10b: lc -b1 -cfist -v -y -oanimtools.o animtools.c
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <graphics/gfx.h>
#include <graphics/gels.h>
#include <graphics/clip.h>
#include <graphics/rastport.h>
#include <graphics/view.h>
#include <graphics/gfxbase.h>
#include "animtools.h"

/* Setup the GELs system.  After this call is made you can use VSprites, Bobs, AnimComps
** and AnimObs.  Note that this links the GelsInfo structure into the RastPort, and calls
** InitGels().  It uses information in your RastPort structure to establish boundary collision
** defaults at the outer edges of the raster.  This routine sets up for everything - collision
** detection and all. You must already have run LoadView before ReadyGelSys is called.
*/
struct GelsInfo *setupGelSys(struct RastPort *rPort, BYTE reserved)
{
struct GelsInfo *gInfo;
struct VSprite  *vsHead;
struct VSprite  *vsTail;

if (NULL != (gInfo = (struct GelsInfo *)AllocMem(sizeof(struct GelsInfo), MEMF_CLEAR)))
        {
        if (NULL != (gInfo->nextLine = (WORD *)AllocMem(sizeof(WORD) * 8, MEMF_CLEAR)))
            {
            if (NULL != (gInfo->lastColor = (WORD **)AllocMem(sizeof(LONG) * 8, MEMF_CLEAR)))
                {
                if (NULL != (gInfo->collHandler = (struct collTable *)
                        AllocMem(sizeof(struct collTable),MEMF_CLEAR)))
                    {
                    if (NULL != (vsHead = (struct VSprite *)
                            AllocMem((LONG)sizeof(struct VSprite), MEMF_CLEAR)))
                        {
                        if (NULL != (vsTail = (struct VSprite *)
                                AllocMem(sizeof(struct VSprite), MEMF_CLEAR)))
                            {
                            gInfo->sprRsrvd   = reserved;
                            /* Set left- and top-most to 1 to better keep items */
                            /* inside the display boundaries.                   */
                            gInfo->leftmost   = gInfo->topmost    = 1;
                            gInfo->rightmost  = (rPort->BitMap->BytesPerRow << 3) - 1;
                            gInfo->bottommost = rPort->BitMap->Rows - 1;
                            rPort->GelsInfo = gInfo;
                            InitGels(vsHead, vsTail, gInfo);
                            return(gInfo);
                            }
                        FreeMem(vsHead, (LONG)sizeof(*vsHead));
                        }
                    FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
                    }
                FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
                }
            FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
            }
        FreeMem(gInfo, (LONG)sizeof(*gInfo));
        }
return(NULL);
}


/* Free all of the stuff allocated by setupGelSys().  Only call this routine if
** setupGelSys() returned successfully.  The GelsInfo structure is the one returned
** by setupGelSys().   It also unlinks the GelsInfo from the RastPort.
*/
```

```
VOID cleanupGelSys(struct GelsInfo *gInfo, struct RastPort *rPort)
{
rPort->GelsInfo = NULL;
FreeMem(gInfo->collHandler, (LONG)sizeof(struct collTable));
FreeMem(gInfo->lastColor, (LONG)sizeof(LONG) * 8);
FreeMem(gInfo->nextLine, (LONG)sizeof(WORD) * 8);
FreeMem(gInfo->gelHead, (LONG)sizeof(struct VSprite));
FreeMem(gInfo->gelTail, (LONG)sizeof(struct VSprite));
FreeMem(gInfo, (LONG)sizeof(*gInfo));
}


/* Create a VSprite from the information given in nVSprite.  Use freeVSprite()
** to free this GEL.
*/
struct VSprite *makeVSprite(NEWVSPRITE *nVSprite)
{
struct VSprite *vsprite;
LONG            line_size;
LONG            plane_size;

line_size = sizeof(WORD) * nVSprite->nvs_WordWidth;
plane_size = line_size * nVSprite->nvs_LineHeight;

if (NULL != (vsprite = (struct VSprite *)AllocMem((LONG)sizeof(struct VSprite), MEMF_CLEAR)))
        {
        if (NULL != (vsprite->BorderLine = (WORD *)AllocMem(line_size, MEMF_CHIP)))
            {
            if (NULL != (vsprite->CollMask = (WORD *)AllocMem(plane_size, MEMF_CHIP)))
                {
                vsprite->Y          = nVSprite->nvs_Y;
                vsprite->X          = nVSprite->nvs_X;
                vsprite->Flags      = nVSprite->nvs_Flags;
                vsprite->Width      = nVSprite->nvs_WordWidth;
                vsprite->Depth      = nVSprite->nvs_ImageDepth;
                vsprite->Height     = nVSprite->nvs_LineHeight;
                vsprite->MeMask     = nVSprite->nvs_MeMask;
                vsprite->HitMask    = nVSprite->nvs_HitMask;
                vsprite->ImageData  = nVSprite->nvs_Image;
                vsprite->SprColors  = nVSprite->nvs_ColorSet;
                vsprite->PlanePick  = vsprite->PlaneOnOff = 0x00;
                InitMasks(vsprite);
                return(vsprite);
                }
            FreeMem(vsprite->BorderLine, line_size);
            }
        FreeMem(vsprite, (LONG)sizeof(*vsprite));
        }
return(NULL);
}


/* Create a Bob from the information given in nBob.  Use freeBob() to free this GEL.
** A VSprite is created for this bob.  This routine properly allocates all double
** buffered information if it is required.
*/
struct Bob *makeBob(NEWBOB *nBob)
{
struct Bob         *bob;
struct VSprite     *vsprite;
NEWVSPRITE          nVSprite ;
LONG                rassize;

rassize = (LONG)sizeof(UWORD) * nBob->nb_WordWidth * nBob->nb_LineHeight * nBob->nb_RasDepth;

if (NULL != (bob = (struct Bob *)AllocMem((LONG)sizeof(struct Bob), MEMF_CLEAR)))
        {
        if (NULL != (bob->SaveBuffer = (WORD *)AllocMem(rassize, MEMF_CHIP)))
            {
            nVSprite.nvs_WordWidth  = nBob->nb_WordWidth;
            nVSprite.nvs_LineHeight = nBob->nb_LineHeight;
            nVSprite.nvs_ImageDepth = nBob->nb_ImageDepth;
            nVSprite.nvs_Image      = nBob->nb_Image;
            nVSprite.nvs_X          = nBob->nb_X;
            nVSprite.nvs_Y          = nBob->nb_Y;
            nVSprite.nvs_ColorSet   = NULL;
```

```
            nVSprite.nvs_Flags       = nBob->nb_BFlags;
            /* Push the values into the NEWVSPRITE structure for use in makeVSprite(). */
            nVSprite.nvs_MeMask      = nBob->nb_MeMask;
            nVSprite.nvs_HitMask     = nBob->nb_HitMask;

            if ((vsprite = makeVSprite(&nVSprite)) != NULL)
                {
                vsprite->PlanePick = nBob->nb_PlanePick;
                vsprite->PlaneOnOff = nBob->nb_PlaneOnOff;
                vsprite->VSBob    = bob;
                bob->BobVSprite   = vsprite;
                bob->ImageShadow = vsprite->CollMask;
                bob->Flags        = 0;
                bob->Before       = NULL;
                bob->After        = NULL;
                bob->BobComp      = NULL;

                if (nBob->nb_DBuf)
                    {
                    if (NULL != (bob->DBuffer = (struct DBufPacket *)
                            AllocMem((LONG)sizeof(struct DBufPacket), MEMF_CLEAR)))
                        {
                        if (NULL != (bob->DBuffer->BufBuffer = (WORD *)AllocMem(rassize, MEMF_CHIP)))
                            return(bob);
                        FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
                        }
                    }
                else
                    {
                    bob->DBuffer = NULL;
                    return(bob);
                    }
                freeVSprite(vsprite);
                }
            FreeMem(bob->SaveBuffer, rassize);
            }
        FreeMem(bob, (LONG)sizeof(*bob));
        }
return(NULL);
}


/*
** Create a Animation Component from the information given in nAnimComp and nBob.  Use
** freeComp() to free this GEL.  makeComp() calls makeBob(), and links the Bob into an AnimComp.
*/
struct AnimComp *makeComp(NEWBOB *nBob, NEWANIMCOMP *nAnimComp)
{
struct Bob      *compBob;
struct AnimComp *aComp;

if ((aComp = AllocMem((LONG)sizeof(struct AnimComp),MEMF_CLEAR)) != NULL)
        {
        if ((compBob = makeBob(nBob)) != NULL)
            {
            compBob->After   = compBob->Before  = NULL;
            compBob->BobComp = aComp;   /* Link 'em up. */
            aComp->AnimBob      = compBob;
            aComp->TimeSet      = nAnimComp->nac_Time; /* Num ticks active. */
            aComp->YTrans       = nAnimComp->nac_Yt; /* Offset rel to HeadOb */
            aComp->XTrans       = nAnimComp->nac_Xt;
            aComp->AnimCRoutine = nAnimComp->nac_Routine;
            aComp->Flags        = nAnimComp->nac_CFlags;
            aComp->Timer        = 0;
            aComp->NextSeq      = aComp->PrevSeq  = NULL;
            aComp->NextComp     = aComp->PrevComp = NULL;
            aComp->HeadOb       = NULL;
            return(aComp);
            }
        FreeMem(aComp, (LONG)sizeof(struct AnimComp));
        }
return(NULL);
}
```

```
/* Create an Animation Sequence from the information given in nAnimSeq and nBob.  Use
** freeSeq() to free this GEL.  This routine creates a linked list of animation components
** which make up the animation sequence.  It links them all up, making a circular list of
** the PrevSeq and NextSeq pointers. That is to say, the first component of the sequences'
** PrevSeq points to the last component; the last component of * the sequences' NextSeq
** points back to the first component.  If dbuf is on, the underlying Bobs will be set up
** for double buffering.  If singleImage is non-zero, the pImages pointer is assumed to
** point to an array of only one image, instead of an array of 'count' images, and all
** Bobs will use the same image.
*/
struct AnimComp *makeSeq(NEWBOB *nBob, NEWANIMSEQ *nAnimSeq)
{
int seq;
struct AnimComp *firstCompInSeq = NULL;
struct AnimComp *seqComp = NULL;
struct AnimComp *lastCompMade = NULL;
LONG image_size;
NEWANIMCOMP nAnimComp;

/* get the initial image.  this is the only image that is used
** if nAnimSeq->nas_SingleImage is non-zero.
*/
nBob->nb_Image = nAnimSeq->nas_Images;
image_size = nBob->nb_LineHeight * nBob->nb_ImageDepth * nBob->nb_WordWidth;

/* for each comp in the sequence */
for (seq = 0; seq < nAnimSeq->nas_Count; seq++)
        {
        nAnimComp.nac_Xt        = *(nAnimSeq->nas_Xt + seq);
        nAnimComp.nac_Yt        = *(nAnimSeq->nas_Yt + seq);
        nAnimComp.nac_Time      = *(nAnimSeq->nas_Times + seq);
        nAnimComp.nac_Routine   = nAnimSeq->nas_Routines[seq];
        nAnimComp.nac_CFlags    = nAnimSeq->nas_CFlags;
        if ((seqComp = makeComp(nBob, &nAnimComp)) == NULL)
            {
            if (firstCompInSeq != NULL)
                freeSeq(firstCompInSeq, (LONG)nBob->nb_RasDepth);
            return(NULL);
            }
        seqComp->HeadOb = nAnimSeq->nas_HeadOb;
        /* Make a note of where the first component is. */
        if (firstCompInSeq == NULL) firstCompInSeq = seqComp;
        /* link the component into the list */
        if (lastCompMade != NULL) lastCompMade->NextSeq = seqComp;
        seqComp->NextSeq = NULL;
        seqComp->PrevSeq = lastCompMade;
        lastCompMade = seqComp;
        /* If nAnimSeq->nas_SingleImage is zero, the image array has nAnimSeq->nas_Count images. */
        if (!nAnimSeq->nas_SingleImage)
            nBob->nb_Image += image_size;
        }
/* On The last component in the sequence, set Next/Prev to make */
/* the linked list a loop of components.                        */
lastCompMade->NextSeq = firstCompInSeq;
firstCompInSeq->PrevSeq = lastCompMade;

return(firstCompInSeq);
}


/* Free the data created by makeVSprite().  Assumes images deallocated elsewhere. */
VOID freeVSprite(struct VSprite *vsprite)
{
LONG    line_size;
LONG    plane_size;

line_size = (LONG)sizeof(WORD) * vsprite->Width;
plane_size = line_size * vsprite->Height;
FreeMem(vsprite->BorderLine, line_size);
FreeMem(vsprite->CollMask, plane_size);
FreeMem(vsprite, (LONG)sizeof(*vsprite));
}
```

```
/* Free the data created by makeBob().  It's important that rasdepth match the depth you */
/* passed to makeBob() when this GEL was made. Assumes images deallocated elsewhere.     */
VOID freeBob(struct Bob *bob, LONG rasdepth)
{
LONG    rassize =  sizeof(UWORD) * bob->BobVSprite->Width * bob->BobVSprite->Height * rasdepth;

if (bob->DBuffer != NULL)
        {
        FreeMem(bob->DBuffer->BufBuffer, rassize);
        FreeMem(bob->DBuffer, (LONG)sizeof(struct DBufPacket));
        }
FreeMem(bob->SaveBuffer, rassize);
freeVSprite(bob->BobVSprite);
FreeMem(bob, (LONG)sizeof(*bob));
}


/* Free the data created by makeComp().  It's important that rasdepth match the depth you */
/* passed to makeComp() when this GEL was made. Assumes images deallocated elsewhere.    */
VOID freeComp(struct AnimComp *myComp, LONG rasdepth)
{
freeBob(myComp->AnimBob, rasdepth);
FreeMem(myComp, (LONG)sizeof(struct AnimComp));
}


/* Free the data created by makeSeq().  Complimentary to makeSeq(), this routine goes through
** the NextSeq pointers and frees the Components.  This routine only goes forward through the
** list, and so it must be passed the first component in the sequence, or the sequence must
** be circular (which is guaranteed if you use makeSeq()).  It's important that rasdepth match
** the depth you passed to makeSeq() when this GEL was made.   Assumes images deallocated elsewhere!
*/
VOID freeSeq(struct AnimComp *headComp, LONG rasdepth)
{
struct AnimComp *curComp;
struct AnimComp *nextComp;

/* Break the NextSeq loop, so we get a NULL at the end of the list. */
headComp->PrevSeq->NextSeq = NULL;

curComp = headComp;         /* get the start of the list */
while (curComp != NULL)
        {
        nextComp = curComp->NextSeq;
        freeComp(curComp, rasdepth);
        curComp = nextComp;
        }
}


/* Free an animation object (list of sequences).  freeOb() goes through the NextComp
** pointers, starting at the AnimObs' HeadComp, and frees every sequence.  It only
** goes forward. It then frees the Object itself.  Assumes images deallocated elsewhere!
*/
VOID freeOb(struct AnimOb *headOb, LONG rasdepth)
{
struct AnimComp *curSeq;
struct AnimComp *nextSeq;

curSeq = headOb->HeadComp;          /* get the start of the list */
while (curSeq != NULL)
        {
        nextSeq = curSeq->NextComp;
        freeSeq(curSeq, rasdepth);
        curSeq = nextSeq;
        }
FreeMem(headOb, sizeof(struct AnimOb));
}
```

Function Reference

The following are brief descriptions of the Amiga's graphics animation functions. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 28-1: Graphics Animation Functions

Animation Function	Description
AddAnimOb()	Add an AnimOb to the linked list of AnimObs.
AddBob()	Add a Bob to the current GEL list.
AddVSprite()	Add a VSprite to the current GEL list.
Animate()	Process every AnimOb in the current animation list.
ChangeSprite()	Change the sprite image pointer.
DoCollision()	Test every GEL in a GEL list for collisions.
DrawGList()	Process the GEL list, queueing VSprites, drawing Bobs.
FreeGBuffers()	Deallocate memory obtained by **GetGBuffers()**.
FreeSprite()	Return sprite for use by others and virtual sprite machine.
GetGBuffers()	Attempt to allocate all buffers of an entire AnimOb.
GetSprite()	Attempt to get a sprite for the simple sprite manager.
InitGels()	Initialize a GEL list; must be called before using GELs.
InitGMasks()	Initialize all of the masks of an AnimOb.
InitMasks()	Initialize the **BorderLine** and **CollMask** masks of a VSprite.
MoveSprite()	Move sprite to a point relative to top of **ViewPort**.
RemBob()	Remove a Bob from the GEL list.
RemIBob()	Immediately remove a Bob from the GEL list and the **RastPort**.
RemVSprite()	Remove a VSprite from the current GEL list.
SetCollision()	Set a pointer to a user collision routine.
SortGList()	Sort the current GEL list, ordering its x,y coordinates.

Chapter 29
GRAPHICS LIBRARY AND TEXT

On the Amiga, rendering text is similar to rendering lines and shapes. The Amiga graphics library provides text functions based around the **RastPort** structure, which makes it easy to intermix graphics and text.

About Amiga Fonts

In order to render text, the Amiga needs to have a graphical representation for each symbol or text character. These individual images are known as glyphs. The Amiga gets each glyph from a font in the system font list. At present, the fonts in the system list contain a bitmap of a specific point size for all the characters and symbols of the font.

Fonts are broken up into different font families. For example, the Amiga's Topaz is a font family. Each font family shares a basic look but can have a variety of styles and point sizes.

The style of a font refers to a minor alteration in the way the plain version of the font's characters are rendered. Currently, the Amiga supports three font styles: **bold**, *italics* and underline (the font's style may also be considered plain when it does not have any of these styles). Although these styles can be inherent to a font, they are normally added algorithmically as text is rendered.

On the Amiga, the point size of a font normally refers to the height of the font in pixels. For example, Topaz-8 is 8 pixels high. Because the size of Amiga pixels varies between display modes, the appearance of a font will also vary between display modes. Future versions of the Amiga OS may measure font size in other units. For example, the standard point in the PostScript page description language normally refers to a point as being a square dot that is 1/72 of an inch on a side. Using a standard measuring unit such as the PostScript point makes it possible to create a WYSIWYG (What You See Is What You Get) display that exactly matches printer or other device output.

When the Amiga first starts up, the only fonts in the system font list are Topaz-8 and Topaz-9, both of which are in ROM. Any other fonts must be loaded from disk or generated somehow. In Amiga operating systems prior to Release 2, additional fonts have to be loaded from disk (usually from the FONTS: directory) using the diskfont.library. For each font size of each font family there is a corresponding bitmap file on disk. If there is no bitmap on disk or in ROM for a specific font size, that font size is not available (if the operating system is 1.3 or earlier).

SYSTEM FONTS IN RELEASE 2

Under Release 2 and later versions of the OS, the system has additional font sources at its disposal. If an application asks the diskfont.library to load a font of a size that has no corresponding bitmap on disk, the library can generate that size font. If diskfont.library can find a smaller or larger version of the font requested, it can scale that font's bitmap to the size needed. Of course, because the library is scaling a bitmap, the quality of the bitmap can degenerate in the scaling process.

A more significant improvement to the diskfont.library is that contains AGFA's Intellifont® engine. As of Release 2.04 (V37) the diskfont.library can utilize AGFA Compugraphic font outlines. The Compugraphic fonts (CG fonts) are mathematical outlines that describe what the font's characters look like. The advantage of the outline fonts is that they can be scaled to any point size without the loss of resolution inherent in bitmap scaling. From the programmers point of view, no extra information is necessary for using the CG fonts, the diskfont.library takes care of all the scaling. Future releases of the OS may bring finer control over the font scaling engine which will allow an application to rotate and shear glyphs.

The Text Function

Amiga text rendering is centered around the graphics.library function **Text()**, which renders text into a rastport:

```
void Text( struct RastPort *myrp, STRPTR mystring, ULONG count );
```

where **myrp** is a pointer to the target rastport, **mystring** is the string to render, and count is the number of characters of **mystring** to render. **Text()** renders at the current rastport position and it takes care of moving the rastport's current X position as it renders each letter. **Text()** only renders text horizontally, so repositioning the rastport's Y position (for example, for a new line) is the responsibility of the application. This is covered in more detail later in this chapter.

Like the other rastport based graphics primitives, most of the text rendering attributes are specified within the **RastPort** structure itself. The current position, the color of the text, and even the font itself are all specified in the **RastPort** structure.

CHOOSING THE FONT

The graphics.library function **SetFont()** changes the rastport's current font:

```
void SetFont( struct RastPort *myrp, struct TextFont *mytf );
```

The parameter **mytf** is a pointer to an open, valid **TextFont** structure. The system uses the **TextFont** structure to keep track of fonts (The **TextFont** structure is discussed in detail later in this chapter). The **OpenFont()** (from graphics.library) and **OpenDiskFont()** (from diskfont.library) functions both return a pointer to a valid **TextFont** structure. The **OpenFont()** function will only open fonts that have already been loaded and are currently in the system list. Normally applications use the **OpenDiskFont()** call instead of **OpenFont()** because **OpenDiskFont()** can load and open fonts from disk as well as open those that are already in the system list.

Here are prototypes for these functions:

```
struct TextFont *OpenDiskFont( struct TextAttr *mytextAttr );
struct TextFont *OpenFont( struct TextAttr *mytextAttr );
```

The **mytextAttr** argument points to a **TextAttr** structure that describes the requested font. The **TextAttr** structure (from *<graphics/text.h>*) looks like this:

```
struct TextAttr {
    STRPTR  ta_Name;          /* name of the font */
    UWORD   ta_YSize;         /* height of the font */
    UBYTE   ta_Style;         /* intrinsic font style */
    UBYTE   ta_Flags;         /* font preferences and flags */
};
```

where **ta_Name** is a string naming the font to open, **ta_YSize** is the point size of the font (normally in pixels), **ta_Style** is a bitfield describing the font style, and **ta_Flags** is a bitfield that further describes characteristics of the font. Note that the name of the font can either be the font name alone (*<font name>.font*) or it can be prepended with a full path. Without a path to the font, if the font is not already loaded into the system list, **OpenDiskFont()** will look in the FONTS: directory for the font file. If there is a path, **OpenDiskFont()** will look in that directory for the font files, allowing the user to put fonts in any directory (although this is discouraged). **OpenFont()** and **OpenDiskFont()** try to find a font that matches your **TextAttr** description. An important thing to remember about **OpenDiskFont()** is that only a process can call it (as opposed to a task). This is primarily because the function has to use dos.library to scan disks for font files.

The font styles for **ta_Style** (from *<graphics/text.h>*) are:

```
FSF_UNDERLINED     The font is underlined
FSF_BOLD           The font is bolded
FSF_ITALIC         The font is italicized
FSF_EXTENDED       The font is extra wide
```

The flags for **ta_Flags** (from *<graphics/text.h>*) are:

```
FPF_ROMFONT       This font is built into the ROM (currently, only Topaz-8 and Topaz-9 are ROM fonts).
FPF_DISKFONT      This font was loaded from disk (with diskfont.library)
FPF_REVPATH       This font is designed to be printed from from right to left (Hebrew is written from
                  right to left)
FPF_TALLDOT       This font was designed for a Hires screen (640x200 NTSC, non-interlaced)
FPF_WIDEDOT       This font was designed for a Lores Interlaced screen (320x400 NTSC)
FPF_PROPORTIONAL  The character widths of this font are not constant
FPF_DESIGNED      This font size was explicitly designed at this size rather than constructed.  If
                  you do not set this bit in your TextAttr, then the system may generate a font
                  for you by scaling an existing ROM or disk font (under V36 and above).
```

For example to open an 11 point bold, italic Topaz font, the code would look something like this:

```
/* pseudotext.c */

void main(void, void)
{
    struct TextAttr myta = {
        "topaz.font"
        11,
        FSF_ITALIC | FSF_BOLD,
        NULL
    };

    struct TextFont *myfont, *oldfont;
    struct RastPort *myrp;
    struct Window   *mywin;
```

```
    . . .
    /* open the graphics and diskfont libraries and whatever else you may need */
    . . .
    if (myfont = OpenDiskFont(&myta))
    {
        /* you would probably set the font of the rastport you are going to use */
        myrp    = mywin->RPort
        oldfont = myrp->Font;
        SetFont(myrp, myfont);

        . . .

        /* perform whatever drawing you need to do */

        . . .

        /* time to clean up.  If the rastport is not exclusively yours,
           you may need to restore the original font or other Rasport values */
        SetFont(myrp, oldfont);
        CloseFont(myfont);
    }

    /* close whatever libraries and other resources you allocated */
}
```

The example above uses the graphics.library's **SetFont()** function to change the rastport's current font. Notice that this example restores the rastport's original font (**myrp->Font**) before exiting. This isn't normally necessary unless some other process assumes the rastport's font (or other drawing attributes) will not change. Intuition does not rely on the window's **RPort.Font** field for rendering or closing the default window font, so applications can change that font without having to restore it.

Prior to Release 2, some applications assumed that any window they opened would always use Topaz-8 without bothering to explicitly set it. Since Topaz-8 was the normal default font before Release 2, this was usually not a problem. However, under Release 2 and later versions of the OS, the user can easily change the default system fonts with the Font Preferences editor. Hence, applications that make assumptions about the size of the default font look terrible under Release 2 (and in some cases are unusable). Program designers should not make assumptions about the system font, and wherever possible, honor the user font preferences. See the "Preferences" chapter of this manual for more information on how to find user preferences.

SETTING THE TEXT DRAWING ATTRIBUTES

In addition to **SetFont()**, there are three rastport control functions that set attributes for text rendering:

```
void SetAPen( struct RastPort *rp, ULONG pen );
void SetBPen( struct RastPort *rp, ULONG pen );
void SetDrMd( struct RastPort *rp, ULONG drawMode );
```

The color of the text depends upon the rastport's current drawing mode and pen colors. You set the draw mode with the **SetDrMd()** function passing it a pointer to a rastport and a drawing mode: JAM1, JAM2, COMPLEMENT or INVERSEID.

If the drawing mode is JAM1, the text will be rendered in the **RastPort.FgPen** color. Wherever there is a set bit in the character's bitmap image, **Text()** will set the corresponding bit in the rastport to the **FgPen** color. This is known as overstrike mode. You set the **FgPen** color with the **SetAPen()** function by passing it a pointer to the rastport and a color number.

If the drawing mode is set to JAM2, **Text()** will place the **FgPen** color as in the JAM1 mode, but it will also set the bits in the rastport to the **RastPort.BgPen** color wherever there is a corresponding cleared bit in the character's bitmap image. Basically, this prints the character themselves in the **FgPen** color and fills in the surrounding parts of the character image with the **BgPen** color. You set the **BgPen** color with the **SetBPen()** function by passing it a pointer to the rastport and a color number.

If the drawing mode is COMPLEMENT, for every bit set in the character's bitmap image, the corresponding bits in the rastport (in all of the rastport's bitplanes) will have their state reversed. cleared bits in the character's bitmap image have no effect on the destination rastport. As with the other drawing modes, the write mask can be used to protect certain bitplanes from being modified (see the graphics primitives chapter for more details).

The JAM1, JAM2, and COMPLEMENT drawing modes are mutually exclusive of each other but each can be modified by the INVERSVID drawing mode. If you combine any of the drawing modes with INVERSVID, the Amiga will reverse the state of all the bits in the source drawing area before writing anything into the rastport.

The idea of using a **RastPort** structure to hold all the rendering attributes is convenient if the rastport's drawing attributes aren't going to change much. This is not the case where several processes need to render into a rastport using very different drawing attributes. An easy way around this problem is to clone the **RastPort**. By making an *exact duplicate* of a **RastPort**, you can change the various rendering parameters of your **RastPort** without effecting other programs that render into the **RastPort** you cloned. Because a **RastPort** only contains a pointer to the rendering area (the bitmap), the original **RastPort** and the cloned **RastPort** both render into the bitmap, but they can use different drawing parameters (font, drawing mode, colors, etc.).

RENDERING THE TEXT

When the **Text()** routine renders text, it renders at the current rastport position along the text's baseline. The baseline is an imaginary line on top of which the text is rendered. Each font has a baseline that is a constant number of pixels from the top of the font's bitmap. For most fonts, parts of some characters are rendered both above and below the baseline (for example, y, g, and j usually have parts above and below the baseline). The part below the baseline is called the descender.

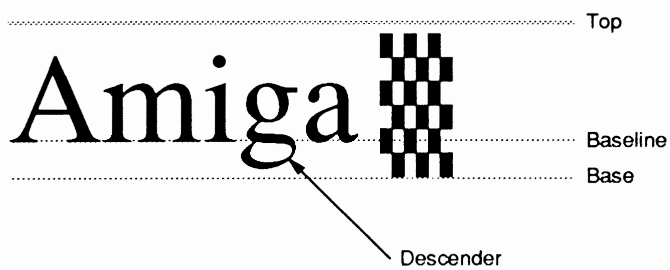


Figure 29-1: Descenders and Baseline of Amiga Fonts

Because **Text()** only increments the rastport's current X position as it renders text horizontally, programs that need to print several lines of text have to take care of moving the current pointer to the beginning of the next line, usually with the graphics.library's **Move()** function:

```
void Move( struct RastPort *rp, LONG x, long y );
```

When moving the current position to the beginning of the next line, an application must make sure it leaves enough space above and below the baseline to prevent characters on different lines from overlapping each other. There are a few fields in the **TextFont** structure returned by **OpenFont()** and **OpenDiskFont()** that are useful for spacing and rendering text:

```
struct TextFont {
    struct Message tf_Message;  /* reply message for font removal */
                                /* font name in LN       |    used in this */
    UWORD   tf_YSize;           /* font height           |    order to best */
    UBYTE   tf_Style;           /* font style            |    match a font */
    UBYTE   tf_Flags;           /* preferences and flags  /    request. */
    UWORD   tf_XSize;           /* nominal font width */
    UWORD   tf_Baseline;        /* distance from the top of char to baseline */
    UWORD   tf_BoldSmear;       /* smear to affect a bold enhancement */

    UWORD   tf_Accessors;       /* access count */

    UBYTE   tf_LoChar;          /* the first character described here */
    UBYTE   tf_HiChar;          /* the last character described here */
    APTR    tf_CharData;        /* the bit character data */

    UWORD   tf_Modulo;          /* the row modulo for the strike font data */
    APTR    tf_CharLoc;         /* ptr to location data for the strike font */
                                /*   2 words: bit offset then size */
    APTR    tf_CharSpace;       /* ptr to words of proportional spacing data */
    APTR    tf_CharKern;        /* ptr to words of kerning data */
};
```

The fields of interest to applications are as follows.

tf_YSize

The "height", in pixels, of this font. None of the characters in this font will be taller than this value.

tf_XSize

This is the character width for monospaced (non-proportional) fonts. The width includes the extra space needed to the left and right of the character to keep the characters from running together.

tf_Baseline

The distance in pixels from the top line of the font to the baseline.

tf_LoChar

This is the first character glyph (the graphical symbol associated with this font) defined in this font. All characters that have ASCII values below this value do not have an associated glyph.

tf_HiChar

This is the last character glyph defined in this font. All characters that have ASCII values above this value do not have an associated glyph. An application can use these values to avoid rendering characters which have no associated glyphs. Any characters without an associated glyph will have the default glyph associated to them. Normally, the default glyph is a rectangle.

To erase text, the graphics.library provides two functions that were specifically designed to clear parts of a rastport based on the dimensions of the current font:

```
void ClearEOL( struct RastPort *rp );
void ClearScreen( struct RastPort *rp );
```

Using the current font, **ClearEOL()** will clear the rest of the current text line from the rastport's current position to the edge of the rastport. **ClearEOL()** was introduced in the Release 2 graphics.library. **ClearScreen()** will clear the rest of the line as **ClearEOL()** does, but it will also clear the rastport below the current line of text.

SETTING THE FONT STYLE

The **OpenFont()** and **OpenDiskFont()** functions both search through the fonts available to them, looking for the font that most closely matches the **TextAttr** structure. If these functions can't find a font that matches exactly, they will open the one with the same name that most closely matches the **TextAttr** structure's **ta_YSize**, **ta_Style**, and **ta_Flags** fields (in that order of preference).

If the font doesn't match your style choice exactly, it is possible to ask the system to alter how it renders the font so it matches the style you need. The rastport contains some flags that tell the system's text rendering functions to algorithmically add styles to characters as they are rendered. Currently, the system can add up to three styles to a font: italics, bold, and underline. The system cannot alter the style of a font if the style is already intrinsic to the font. For example, it is not possible to add (or remove) the bold styling to a font if the font was designed to be bolded. There are two graphics.library functions that deal with software font style setting:

```
ULONG AskSoftStyle( struct RastPort *rp );
ULONG SetSoftStyle( struct RastPort *rp, ULONG newstyle, ULONG enable );
```

The **AskSoftStyle()** function returns a bitmask of the style bits available to the rastport's current font. The style bits are the same ones used by the **TextAttr**'s **ta_Style** field (from *<graphics/text.h>*). **SetSoftStyle()** changes the rastport's current software style setting according to the style bits set in the **newstyle** field (from the function prototype above).

SetSoftStyle() pays attention only to the bits of **newstyle** that have the corresponding bit in the enable field set as well. This function returns the style, which is the combined result of previous soft style selection, the effect of this function, and the style inherent in the set font. The following code fragment turns on the algorithmic font attributes for the rastport (**myrastport**) based on those style attributes that were requested in the **OpenDiskFont()** call (**mytextattr.ta_Style**) and *not* inherent in the font.

```
/* Set the font and add software styling to the text if I asked for a
   style in OpenFont() and didn't get it.  Because most Amiga fonts do
   not have styling built into them (with the exception of the CG outline
   fonts), if the user selected some kind of styling for the text, it
   will have to be added algorithmically by calling SetSoftStyle().
*/
if (myfont = OpenDiskFont(mytextattr))
{
        SetFont(myrastport, myfont);
        SetSoftStyle(myrastport,
                     mytextattr.ta_Style ^ myfont->tf_Style,
                     (FSF_BOLD | FSF_UNDERLINED | FSF_ITALIC));
        CloseFont(myfont);
}
```

Does the Text Fit?

The **Text()** function renders its text on a single horizontal line without considering whether or not the text it renders will actually fit in the visible portion of the display area. Although for some applications this behavior is acceptable, other applications, for example a word processor, need to render all of their text where the user can see it. These applications need to measure the display area to determine how much text can fit along a given baseline. The graphics.library contains several functions that perform some of the necessary measurements:

```
WORD TextLength( struct RastPort *my_rp, STRPTR mystring, ULONG mycount );

void TextExtent( struct RastPort *my_rp, STRPTR mystring, LONG mycount,
                 struct TextExtent *textExtent );

void FontExtent( struct TextFont *font, struct TextExtent *fontExtent );

ULONG TextFit   ( struct RastPort *rp, STRPTR mystring, ULONG strLen,
                 struct TextExtent *textExtent, struct TextExtent *constrainingExtent,
                 LONG strDirection, ULONG constrainingBitWidth,
                 ULONG constrainingBitHeight );
```

The **TextLength()** function is intended to mimic the **Text()** function without rendering the text. Using the exact same parameters as the **Text()** function, **TextLength()** returns the change in **my_rp**'s current X position (**my_rp.cp_x**) that would result if the text had been rendered using the **Text()** function. As in **Text()**, the **mycount** parameter tells how many characters of mystring to measure.

Some fonts have characters that intrinsically render outside of the normal rectangular bounds. This can result for example, from the Amiga's version of kerning (which is discussed later in this chapter) or from algorithmic italicizing. In such cases, **TextLength()** is insufficient for determining whether a text string can fit within a given rectangular bounds.

The **TextExtent()** function offers a more complete measurement of a string than the **TextLength()** function. **TextExtent()**, which was introduced in Release 2, fills in the **TextExtent** structure passed to it based on the current rendering settings in **my_rp**.

The **TextExtent** structure *<graphics/text.h>*) supplies the dimensions of **mystring**'s bounding box:

```
struct TextExtent {
    UWORD   te_Width;           /* same as TextLength */
    UWORD   te_Height;          /* same as tf_YSize   */
    struct Rectangle te_Extent; /* relative to CP     */
};
```

The **Rectangle** structure (from *<graphics/gfx.h>*):

```
struct Rectangle
{
    WORD   MinX,MinY;
    WORD   MaxX,MaxY;
};
```

TextExtent() fills in the **TextExtent** structure as follows:

te_Width	the same value returned by **TextLength()**.
te_Height	the font's Y size.
te_Extent.MinX	the pixel offset from the rastport's current X position to the left side of the bounding box defined by the rectangle structure. Normally, this is zero.
te_Extent.MinY	the distance in pixels from the baseline to the top of the bounding box.
te_Extent.MaxX	the pixel offset from the rastport's current X position to the right side of the bounding box. Normally, this is **te_Width** - 1.
te_Extent.MaxY	the distance from the baseline to the bottom of the bounding box.

The **FontExtent()** function is similar to the **TextExtent()** function. It fills in a **TextExtent()** structure that describes the bounding box of the largest possible single character in a particular open font, including the effects of kerning. Because the **TextExtent()** function looks at an open **DiskFont** structure rather than a rastport to figure out values of the **TextExtent** structure, it cannot consider the effects of algorithmic styling. Like **TextExtent()**, **FontExtent()** was introduced in Release 2, so it is not available under the 1.3 or earlier OS releases.

The **TextFit()** function looks at a string and returns the number of characters of the string that will fit into a given rectangular bounds. **TextFit()** takes the current rastport rendering settings into consideration when measuring the text. Its parameters (from the prototype above) are:

my_rp	tells which rastport to get the rendering attributes from
mystring	the string to ''fit''
strLen	number of characters of **mystring** to ''fit''
constrainingExtent	a **TextExtent** structure describing the bounding box in which to ''fit'' **mystring**
strDirection	the offset to add to the string pointer to get to the next character in **mystring** (can be negative)
constrainingBitWidth	an alternative way to specify the width of the bounding box in which to ''fit'' **mystring**
constrainingBitHeight	an alternative way to specify the height of the bounding box in which to ''fit'' **mystring**

TextFit() will only pay attention to the **constrainingBitWidth** and **constrainingBitHeight** fields if **constrainingExtent** is NULL.

TEXT MEASURING EXAMPLE

The following example, measuretext.c, opens a window on the default public screen and renders the contents of an ASCII file into the window. It uses **TextFit()** to measure how much of a line of text will fit across the window. If the entire line doesn't fit, measuretext will wrap the remainder of the line into the rows that follow. This example makes use of an ASL font requester, letting the user choose the font, style, size, drawing mode, and color.

```
;/* MeasureText - Execute me to compile me with Lattice 5.10a
LC -b0 -cfistq -v -y -j73 MeasureText.c
Blink FROM LIB:c.o, MeasureText.o TO MeasureText LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;
*/
#define INTUITION_IOBSOLETE_H
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <graphics/text.h>
#include <graphics/rastport.h>
#include <intuition/intuition.h>
#include <exec/libraries.h>

#include <clib/alib_stdio_protos.h>
#include <clib/graphics_protos.h>
#include <clib/intuition_protos.h>
#include <clib/diskfont_protos.h>
#include <clib/dos_protos.h>
#include <clib/exec_protos.h>
#include <clib/asl_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

#define BUFSIZE 32768

UBYTE *vers = "\0$VER: MeasureText 37.1";

UBYTE buffer[BUFSIZE];

void MainLoop(void);
void EOP(void);

struct Library *IntuitionBase, *GfxBase, *DiskfontBase, *AslBase;
BPTR myfile;
UWORD wtbarheight;
struct FontRequester *fr;
struct TextFont *myfont;
struct Window *w;
struct RastPort *myrp;
struct Task *mytask;

void main(int argc, char **argv)
{
  struct TextAttr myta;

  if (argc == 2)
  {
    if (myfile = Open(argv[1], MODE_OLDFILE))                  /* Open the file to print out. */
    {
      if (DiskfontBase = OpenLibrary("diskfont.library", 37L))        /* Open the libraries. */
      {
        if (IntuitionBase = OpenLibrary("intuition.library", 37L))
        {
          if (GfxBase = OpenLibrary("graphics.library", 37L))
          {
            if (AslBase = OpenLibrary("asl.library", 37L))
            {
              if (fr = (struct FontRequester *)              /* Open an ASL font requester */
                      AllocAslRequestTags(ASL_FontRequest,
                                  /* Supply initial values for requester */
                                  ASL_FontName, (ULONG)"topaz.font",
                                  ASL_FontHeight, 11L,
                                  ASL_FontStyqles, FSF_BOLD | FSF_ITALIC,
                                  ASL_FrontPen,  0x01L,
                                  ASL_BackPen,   0x00L,

                                   /* Give us all the gadgetry */
                                  ASL_FuncFlags, FONF_FRONTCOLOR | FONF_BACKCOLOR |
                                                 FONF_DRAWMODE | FONF_STYLES,
                                  TAG_DONE))
```

```
            {
              /* Pop up the requester */
              if (AslRequest(fr, 0L))
              {
                myta.ta_Name       = fr->fo_Attr.ta_Name;         /* extract the font and */
                myta.ta_YSize      = fr->fo_Attr.ta_YSize;        /* display attributes   */
                myta.ta_Style      = fr->fo_Attr.ta_Style;        /* from the FontRequest */
                myta.ta_Flags      = fr->fo_Attr.ta_Flags;        /* structure.           */

                if (myfont = OpenDiskFont(&myta))
                {
                  if (w = OpenWindowTags(NULL,WA_SizeGadget,  TRUE,
                                              WA_MinWidth,    200,
                                              WA_MinHeight,   200,
                                              WA_DragBar,     TRUE,
                                              WA_DepthGadget, TRUE,
                                              WA_Title,       (ULONG)argv[1],
                                              TAG_DONE))
                  {
                    myrp = w->RPort;
                    /* figure out where the baseline of the uppermost line should be. */
                    wtbarheight = w->WScreen->BarHeight + myfont->tf_Baseline + 2;

                    /* Set the font and add software styling to the text if I asked for it */
                    /* in OpenFont() and didn't get it.  Because most Amiga fonts do not   */
                    /* have styling built into them (with the exception of the CG outline  */
                    /* fonts), if the user selected some kind of styling for the text, it  */
                    /* will to be added algorithmically by calling SetSoftStyle().         */

                    SetFont(myrp, myfont);
                    SetSoftStyle(myrp,   myta.ta_Style ^ myfont->tf_Style,
                                 (FSF_BOLD | FSF_UNDERLINED | FSF_ITALIC));
                    SetDrMd(myrp, fr->fo_DrawMode);
                    SetAPen(myrp, fr->fo_FrontPen);
                    SetBPen(myrp, fr->fo_BackPen);
                    Move(myrp, w->WScreen->WBorLeft, wtbarheight);
                    mytask = FindTask(NULL);

                    MainLoop();

                    Delay(25);                   /* short delay to give user a chance to */
                    CloseWindow(w);              /* see the text before it goes away.    */
                  }
                  CloseFont(myfont);
                }
              }
              else
                VPrintf("Request Cancelled\n", NULL);
              FreeAslRequest(fr);
            }
            CloseLibrary(AslBase);
          }
          CloseLibrary(GfxBase);
        }
        CloseLibrary(IntuitionBase);
      }
      CloseLibrary(DiskfontBase);
    }
    Close(myfile);
  }
}
else
  VPrintf("template: MeasureText <file name>\n", NULL);
}


void MainLoop(void)
{
  struct TextExtent resulttextent;
  LONG fit, actual, count, printable, crrts;
  BOOL aok = TRUE;

  while (((actual = Read(myfile, buffer, BUFSIZE)) > 0) && aok)  /* while there's something to */
  {                                                               /* read, fill the buffer.     */
    count = 0;
```

```
    while(count < actual)
    {
      crrts = 0;

      while ( ((buffer[count] < myfont->tf_LoChar) ||     /* skip non-printable characters, but */
               (buffer[count] > myfont->tf_HiChar)) &&   /* account for newline characters.    */
               (count < actual) )
      {
        if (buffer[count] == '\012') crrts++; /* is this character a newline?  if it is, bump */
        count++;                               /* up the newline count.                        */
      }

      if (crrts > 0)                  /* if there where any newlines, be sure to display them. */
      {
        Move(myrp, w->BorderLeft, myrp->cp_y + (crrts * (myfont->tf_YSize + 1)));
        EOP();                                         /* did we go past the end of the page? */
      }

      printable = count;
      while ( (buffer[printable] >= myfont->tf_LoChar) &&      /* find the next non-printables */
              (buffer[printable] <= myfont->tf_HiChar) &&
              (printable < actual) )
      {
        printable++;
      }                                   /* print the string of printable characters wrapping  */
      while (count < printable)           /* lines to the beginning of the next line as needed. */
      {
        /* how many characters in the current string of printable characters will fit */
        /* between the rastport's current X position and the edge of the window?      */
        fit = TextFit( myrp,               &(buffer[count]),
                       (printable - count), &resulttextent,
                       NULL,               1,
                       (w->Width  - (myrp->cp_x + w->BorderLeft + w->BorderRight)),
                       myfont->tf_YSize + 1  );
        if ( fit == 0 )
        {
            /* nothing else fits on this line, need to wrap to the next line.        */
            Move(myrp, w->BorderLeft, myrp->cp_y + myfont->tf_YSize + 1);
        }
        else
        {
           Text(myrp, &(buffer[count]), fit);
           count += fit;
        }
        EOP();
      }

      if (mytask->tc_SigRecvd & SIGBREAKF_CTRL_C)        /* did the user hit CTRL-C (the shell */
      {                                                  /* window has to receive the CTRL-C)? */
        aok = FALSE;
        VPrintf("Ctrl-C Break\n", NULL);
        count = BUFSIZE + 1;
      }
    }
  }
  if (actual < 0)
    VPrintf("Error while reading\n", NULL);
}


void EOP(void)
{
    if (myrp->cp_y > (w->Height - (w->BorderBottom + 2))) /* If we reached page bottom, clear the */
    {                                                     /* rastport and move back to the top.   */
        Delay(25);

        SetAPen(myrp, 0);
        RectFill(myrp, (LONG)w->BorderLeft, (LONG)w->BorderTop, w->Width - (w->BorderRight + 1),
                 w->Height - (w->BorderBottom + 1) );
        SetAPen(myrp, 1);
        Move(myrp, w->BorderLeft + 1, wtbarheight);
        SetAPen(myrp, fr->fo_FrontPen);
    }
}
```

Font Scaling and Aspect Ratio

The Release 2 OS offers a significant improvement over the Amiga's previous font resources: it now has the ability to scale fonts to new sizes and dimensions. This means, if the diskfont.library can't find the font size an application requests, it can create a new bitmap font by scaling the bitmap of a different size font in the same font family. The 2.04 (V37) release of the OS improved upon the diskfont.library's font scaling ability so the Amiga now can utilize AGFA Compugraphic outline fonts, yielding scalable fonts that don't have the exaggerated jagged edges inherent in bitmap scaling.

The best thing about the Amiga's font scaling is that its addition to the system is completely invisible to an application program. Because the diskfont.library takes care of all the font scaling, any program that uses **OpenDiskFont()** to open a font can have scalable fonts available to it. For simple scaling, the programming interface is the same using Release 2 as it was under 1.3.

However, there is one feature of the Release 2 diskfont.library that the 1.3 programming interface cannot handle. When scaling a font (either from an outline or from another bitmap), the Release 2 diskfont.library can adjust the width of a font's glyphs according to an aspect ratio passed to **OpenDiskFont()**. A font glyph is the graphical representations associated with each symbol or character of a font.

The aspect ratio refers to the shape of the pixels that make up the bitmap that diskfont.library creates when it scales a font. This ratio is the width of a pixel to the height of the pixel (**XWidth/ YWidth**). The diskfont.library uses this ratio to figure out how wide to make the font glyphs so that the look of a font's glyphs will be the same on display modes with very different aspect ratios.

To add this new feature, several changes to the OS were necessary:

1) The OS needed to be able to store an aspect ratio for any font loaded into the system list.

2) The structures that diskfont.library uses to store bitmap fonts on disk had to be updated so they can store the aspect ratio a font was designed for.

3) The way in which an application requests fonts from diskfont.library had to be altered so that an application could ask for a specific aspect ratio.

For the diskfont.library to be able to scale a font to a new aspect ratio, it needs to know what the font's current aspect ratio is. The Amiga gets the aspect ratio of a font currently in the system list from an extension to the **TextFont** structure called (oddly enough) **TextFontExtension**. Under Release 2, when the system opens a new font (and there is sufficient memory), it creates this extension.

A font's **TextFont** structure contains a pointer to its associated **TextFontExtension**. While the font is opened, the **TextFont**'s **tf_Message.mn_ReplyPort** field points to a font's **TextFontExtension**. The *<graphics/text.h>* include file #defines **tf_Message.mn_ReplyPort** as **tf_Extension**.

The **TextFontExtension** structure contains only one field of interest: a pointer to a tag list associated with this font:

```
struct TagItem *tfe_Tags;            /* Text Tags for the font */
```

If a font has an aspect ratio associated with it, the OS stores the aspect ratio as a tag/value pair in the **tfe_Tags** tag list.

The TA_DeviceDPI tag holds a font's aspect ratio. The data portion of the TA_DeviceDPI tag contains an X DPI (dots per inch) value in its upper word and a Y DPI value in its lower word. These values are unsigned words (UWORD). At present, these values do not necessarily reflect the font's true X and Y DPI, so their specific values are not relevant. At present, only the ratio of the X aspect to the Y aspect is important (more on this later in the article).

Notice that the X and Y DPI values are not aspect values. The X and Y aspect values are the reciprocals of the X and Y DPI values, respectively:

```
XDPI = 1/XAspect
YDPI = 1/YAspect
```

so, the aspect ratio is *YDPI/XDPI*, not *XDPI/YDPI*.

Before accessing the tag list, an application should make sure that this font has a corresponding **TextFontExtension**. The **ExtendFont()** function will return a value of TRUE if this font already has an extension or **ExtendFont()** was able to create an extension for this font.

The Amiga has a place to store a font's X and Y DPI values once the font is loaded into memory, but where do these X and Y values come from? A font's X and Y DPI values can come from several sources. The X and Y DPI can come from a font's disk-based representation, or it can be set by the programmer.

For the traditional Amiga bitmap fonts, in order to store the X and Y DPI values in a bitmap font's ".font" file, the structures that make up the ".font" file had to be expanded slightly. See the discussion of the **FontContentsHeader** structure in the "Composition of a Bitmap Font on Disk" section later in this chapter for more information. Currently, out of all the system standard bitmap fonts (those loaded from bitmaps on disk or ROM, not scaled from a bitmap or outline), only one has a built in aspect ratio: Topaz-9.

For the Compugraphic outline fonts, the X and Y DPI values are built into the font outline. Because the format of the Compugraphic outline fonts is proprietary, information about their layout is available only from AGFA Compugraphic. For most people, the format of the outline fonts is not important, as the diskfont.library handles converting the fonts to an Amiga-usable form.

The other place where a font can get an aspect ratio is an application. When an application opens a font with **OpenDiskFont()**, it can supply the TA_DeviceDPI tag that the diskfont.library uses to scale (if necessary) the font according to the aspect ratio. To do so, an application has to pass **OpenDiskFont()** an extended version of the **TextAttr** structure called the **TTextAttr**:

```
struct TTextAttr {
    STRPTR  tta_Name;            /* name of the font          */
    UWORD   tta_YSize;           /* height of the font        */
    UBYTE   tta_Style;           /* intrinsic font style      */
    UBYTE   tta_Flags;           /* font preferences and flags */
    struct TagItem *tta_Tags;    /* extended attributes       */
};
```

The **TextAttr** and the **TTextAttr** are identical except that the **tta_Tags field** points to a tag list where you place your TA_DeviceDPI tag. To tell **OpenDiskFont()** that it has a **TTextAttr** structure rather than a **TextAttr** structure, set the FSF_TAGGED bit in the **tta_Style** field.

For example, to ask for Topaz-9 scaled to an aspect ratio of 75 to 50 the code might look something like this:

```
#define MYXDPI (75L << 16)
#define MYYDPI (50L)

struct TTextAttr mytta = {
    "topaz.font", 9,
    FSF_TAGGED,   0, NULL
};

struct TagItem tagitem[2];
struct TextFont *myfont;
ULONG dpivalue;

 . . .

tagitem[0].ti_tag = TA_DeviceDPI;
tagitem[0].ti_Data = MYXDPI | MYYDPI;
tagitem[1].ti_tag = TAG_END;
mytta.tta_tags = tagitem;

 . . .

if (myfont = OpenDiskFont(&mytta))
{
    dpi = GetTagData(TA_DeviceDPI,
                     0L,
                     ((struct TextFontExtension *)(myfont->tf_Extension))->tfe_Tags);
    if (dpi) printf("XDPI = %d    YDPI = %d\n",
                    ((dpi & 0xFFFF0000)>>16),
                    (dpi & 0x0000FFFF));
          . . .                        /* Blah Blah print blah */

    CloseFont(myfont);
}
```

Some Things to Look Out For

One misleading thing about the TA_DeviceDPI tag is that its name implies that the diskfont.library is going to scale the font glyphs according to an actual DPI (dots per inch) value. As far as scaling is concerned, this tag serves only as a way to specify the aspect ratio, so the actual values of the X and Y elements are not important, just the ratio of one to the other. A font glyph will look the same if the ratio is 2:1 or 200:100 as these two ratios are equal.

To makes things a little more complicated, when diskfont.library scales a bitmap font using an aspect ratio, the X and Y DPI values that the OS stores in the font's **TextFontExtension** are identical to the X and Y DPI values passed in the TA_DeviceDPI tag. This means the system can associate an X and Y DPI value to an open font size that is very different from the font size's actual X and Y DPI. For this reason, applications should not use these values as real DPI values. Instead, only use them to calculate a ratio.

For the Compugraphic outline fonts, things are a little different. The X and Y DPI values are built into the font outline and reflect a true X and Y DPI. When the diskfont.library creates a font from an outline, scaling it according to an application-supplied aspect ratio, diskfont.library does not change the Y DPI setting. Instead, it calculates a new X DPI based on the font's Y DPI value and the aspect ratio passed in the TA_DeviceDPI tag. It does this because the Amiga thinks of a font size as being a height in pixels. If an application was able to change the true Y DPI of a font, the diskfont.library would end up creating font sizes that were much larger or smaller than the **YSize** the application asked for. If an application needs to scale a font according to height as well as width, the application can adjust the value of the **YSize** it asks for in the **TTextAttr**.

As mentioned earlier, almost all of the system standard bitmap fonts do not have a built in aspect ratio. This means that if an application loads one of these bitmap fonts without supplying an aspect ratio, the system will not put a TA_DeviceDPI tag in the font's **TextFontExtension**: the font will not have an aspect ratio. If a font size that is already in the system font list does not have an associated X and Y DPI, the diskfont.library cannot create a new font of the same size with a different aspect ratio.

The reason for this is the diskfont.library cannot tell the difference between two instances of the same font size where one has an aspect ratio and the other does not. Because diskfont.library cannot see this difference, when an application asks, for example, for Topaz-8 with an aspect ratio of 2:1, **OpenDiskFont()** first looks through the system list to see if that font is loaded. **OpenDiskFont()** happens to find the ROM font Topaz-8 in the system font list, which has no X and Y DPI. Because it cannot see the difference, diskfont.library thinks it has found what it was looking for, so it does not create a new Topaz-8 with an aspect ratio of 2:1, and instead opens the Topaz-8 with no aspect ratio.

This also causes problems for programs that do not ask for a specific aspect ratio. When an application asks for a font size without specifying an aspect ratio, **OpenDiskFont()** will not consider the aspect ratios of fonts in the system font list when it is looking for a matching font. If a font of the same font and style is already in the system font list, even though it may have a wildly distorted aspect ratio, **OpenDiskFont()** will return the font already in the system rather than creating a new one.

FONT ASPECT RATIO EXAMPLE

The following example, cliptext.c, renders the contents of a text file to a Workbench window. This example gets the new aspect ratio for a font by asking the display database what the aspect ratio of the current display mode is.

```
;/* cliptext.c - Execute me to compile me with Lattice 5.10a
LC -cfistq -v -y -j73 cliptext.c
Blink FROM LIB:c.o,cliptext.o TO cliptext LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;*/

#include <exec/types.h>
#include <dos/rdargs.h>
#include <dos/dosextens.h>
#include <intuition/intuition.h>
#include <graphics/text.h>
#include <graphics/displayinfo.h>
#include <graphics/regions.h>
#include <graphics/gfx.h>
#include <libraries/diskfont.h>
#include <libraries/diskfonttag.h>
#include <utility/tagitem.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/layers_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/diskfont_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\0$VER: cliptext 37.2";

#define BUFSIZE         4096
#define FONT_NAME       0
#define FONT_SIZE       1
#define FILE_NAME       2
#define JAM_MODE        3
```

```
#define XASP            4
#define YASP            5
#define NUM_ARGS        6
#define DEFAULTFONTSIZE 11L
#define DEFAULTJAMMODE  0L
#define DEFAULTXASP     0L
#define DEFAULTYASP     0L

void MainLoop(void);

LONG args[NUM_ARGS];
struct TagItem tagitem[2];
UBYTE buffer[BUFSIZE];
BPTR myfile;
struct Library *DiskfontBase, *IntuitionBase, *LayersBase, *GfxBase;
struct IntuiMessage *mymsg;
struct DrawInfo *mydrawinfo;
struct Window *mywin;
struct RastPort *myrp;
struct TTextAttr myta;
struct TextFont *myfont;
struct Rectangle myrectangle;
struct Region *new_region;

void main(int argc, char **argv)
{
  struct RDArgs *myrda;
  struct DisplayInfo mydi;
  ULONG mymodeid;

  LONG mydefaultfontsize = DEFAULTFONTSIZE;
  LONG mydefaultJAMMode = DEFAULTJAMMODE;
  LONG mydefaultXASP = 0L;
  LONG mydefaultYASP = 0L;
  args[FONT_NAME] = (LONG)"topaz.font";
  args[FONT_SIZE] = (LONG)&mydefaultfontsize;
  args[FILE_NAME] = (LONG)"s:startup-sequence";
  args[JAM_MODE]  = (LONG)&mydefaultJAMMode;
  args[XASP]      = (LONG)&mydefaultXASP;
  args[YASP]      = (LONG)&mydefaultYASP;

  /* dos.library standard command line parsing--See the dos.library Autodoc for details */
  if (myrda = ReadArgs("FontName,FontSize/N,FileName,Jam/N,XASP/N,YASP/N\n", args, NULL))
  {
    if (myfile = Open((UBYTE *)args[FILE_NAME], MODE_OLDFILE) )   /* Open the file to display. */
    {
      if (DiskfontBase = OpenLibrary("diskfont.library", 37L))         /* Open the libraries. */
      {
        if (IntuitionBase = OpenLibrary("intuition.library", 37L))
        {
          if (GfxBase = OpenLibrary("graphics.library", 37L))
          {
            if (LayersBase = OpenLibrary("layers.library", 37L))
            {
              if (mywin = OpenWindowTags(NULL,                            /* Open that window. */
                     WA_MinWidth,     100,     /* This application wants to hear about three   */
                     WA_MinHeight,    100,     /* things: 1) When the user clicks the window's */
                     WA_SmartRefresh, TRUE,    /* close gadget, 2) when the user starts to     */
                     WA_SizeGadget,   TRUE,    /* resize the window, 3) and when the user has  */
                     WA_CloseGadget,  TRUE,    /* finished resizing the window.                */
                     WA_IDCMP,       IDCMP_CLOSEWINDOW | IDCMP_NEWSIZE | IDCMP_SIZEVERIFY,
                     WA_DragBar,     TRUE,
                     WA_DepthGadget, TRUE,
                     WA_Title,       (ULONG)args[FILE_NAME],
                     TAG_END))
              {
                tagitem[0].ti_Tag = OT_DeviceDPI;

                /* See if there is a non-zero value in the XASP or YASP fields. Diskfont.library */
                /* will get a divide by zero GURU if you give it a zero XDPI or YDPI value.      */

                /* if there is a zero value in one of them... */
                if (  ( (*(ULONG *)args[XASP]) == 0) || ( (*(ULONG *)args[YASP]) == 0)  )
                {
                  /* ...then use the aspect ratio of the current display as a default... */
                  mymodeid = GetVPModeID(&(mywin->WScreen->ViewPort));
```

```
          if (GetDisplayInfoData( NULL, (UBYTE *)&mydi,
                              sizeof(struct DisplayInfo), DTAG_DISP, mymodeid))
          {
            mydefaultXASP = mydi.Resolution.x;
            mydefaultYASP = mydi.Resolution.y;
            printf("XAsp = %ld    YAsp = %ld\n", mydefaultXASP, mydefaultYASP);
            /* Notice that the X and Y get _swapped_ to keep the look of the    */
            /* font glyphs the same using screens with different aspect ratios. */
            args[YASP]    = (LONG)&mydefaultXASP;
            args[XASP]    = (LONG)&mydefaultYASP;
          }
          else /* ...unless something is preventing us from getting the screen  */
               /* screens resolution.  In that case, forget about the DPI tag.  */
               tagitem[0].ti_Tag = TAG_END;
        }
        /* Here we have to put the X and Y DPI into the OT_DeviceDPI tags data field.  */
        /* THESE ARE NOT REAL X AND Y DPI VALUES FOR THIS FONT OR DISPLAY. They only   */
        /* serve to supply the diskfont.library with values to calculate the aspect    */
        /* ratio.  The X value gets stored in the upper word of the tag value and the Y*/
        /* DPI gets stored in the lower word.  Because ReadArgs() stores the _address_ */
        /* of integers it gets from the command line, you have to dereference the      */
        /* pointer it puts into the argument array, which results in some ugly casting.*/

        tagitem[0].ti_Data = (ULONG)( ( (UWORD) *( (ULONG *)args[XASP] ) << 16) |
                                       ((UWORD) *( (ULONG *)args[YASP]) ) );
        tagitem[1].ti_Tag = TAG_END;

        myta.tta_Name = (STRPTR)args[FONT_NAME];          /* Set up the TTextAttr     */
        myta.tta_YSize = *((LONG *)args[FONT_SIZE]);      /* structure to match the   */
        myta.tta_Style = FSF_TAGGED;                      /* font the user requested. */
        myta.tta_Flags = 0L;
        myta.tta_Tags = tagitem;

        if (myfont = OpenDiskFont(&myta))         /* open that font */
        {
         /* This is for the layers.library clipping region that gets attached to the   */
         /* window.  This prevents the application from unnecessarily rendering beyond */
          myrectangle.MinX = mywin->BorderLeft; /* the bounds of the inner part of     */
          myrectangle.MinY = mywin->BorderTop;  /* the window.  For now, you can       */
          myrectangle.MaxX = mywin->Width -     /* ignore the layers stuff if you are  */
               (mywin->BorderRight + 1);        /* just interested in learning about   */
          myrectangle.MaxY = mywin->Height -    /* using text.  For more information   */
               (mywin->BorderBottom + 1);       /* on clipping regions and layers, see */
                                                /* the Layers chapter of this manual.  */

          if (new_region = NewRegion())                             /* more layers stuff */
          {
            if (OrRectRegion(new_region, &myrectangle));      /* Even more layers stuff */
            {
              InstallClipRegion(mywin->WLayer, new_region);
              /* Obtain a pointer to the window's rastport and set up some of the      */
              myrp = mywin->RPort;     /* rastport attributes. This example obtains the */
              SetFont(myrp, myfont);   /* text pen for the window's screen using        */
              if (mydrawinfo =GetScreenDrawInfo(mywin->WScreen)) /* GetScreenDrawInfo() */
              {
                SetAPen(myrp, mydrawinfo->dri_Pens[TEXTPEN]);
                FreeScreenDrawInfo(mywin->WScreen, mydrawinfo);
              }
              SetDrMd(myrp, (BYTE)(*((LONG *)args[JAM_MODE])));

              MainLoop();
            }
            DisposeRegion(new_region);
          }
          CloseFont(myfont);
        }
        CloseWindow(mywin);
      }
      CloseLibrary(LayersBase);
    }
    CloseLibrary(GfxBase);
  }
  CloseLibrary(IntuitionBase);
 }
 CloseLibrary(DiskfontBase);
}
```

```
      Close(myfile);
    }
    FreeArgs(myrda);
  }
  else VPrintf("Error parsing arguments\n", NULL);
}


void MainLoop(void)
{
    LONG count, actual, position;
    BOOL aok = TRUE, waitfornewsize = FALSE;
    struct Task *mytask;

    mytask = FindTask(NULL);
    Move(myrp, mywin->BorderLeft + 1, mywin->BorderTop + myfont->tf_YSize + 1);

    while (((actual = Read(myfile, buffer, BUFSIZE)) > 0) && aok) /* While there's something   */
    {                                                              /* to read, fill the buffer. */
        position = 0;
        count = 0;

        while (position <= actual)
        {
           if (!(waitfornewsize))
           {
               while ( ((buffer[count] >= myfont->tf_LoChar) &&
                       (buffer[count] <= myfont->tf_HiChar)) && (count <= actual) )
                   count++;

               Text(myrp, &(buffer[position]), (count)-position);

               while ( ((buffer[count] < myfont->tf_LoChar) ||
                       (buffer[count] > myfont->tf_HiChar)) && (count <= actual) )
               {
                   if (buffer[count] == 0x0A)
                       Move(myrp, mywin->BorderLeft, myrp->cp_y + myfont->tf_YSize + 1);
                   count++;
               }
               position = count;
           }
           else WaitPort(mywin->UserPort);

           while (mymsg = (struct IntuiMessage *)GetMsg(mywin->UserPort))
           {
               if (mymsg->Class == IDCMP_CLOSEWINDOW)     /* The user clicked the close gadget */
               {
                   aok = FALSE;
                   position = actual + 1;
                   ReplyMsg((struct Message *)mymsg);
               }                                              /* The user picked up the */
               else if (mymsg->Class == IDCMP_SIZEVERIFY)     /* window's sizing gadget */
               {
                  /* When the user has picked up the window's sizing gadget when the      */
                  /* IDCMP_SIZEVERIFY flag is set, the application has to reply to this message*/
                  /* to tell Intuition to allow the user to move the sizing gadget and resize */
                  /* the window.  The reason for using this here is because the user can resize*/
                  /* the window while cliptext.c is rendering text to the window. Cliptext.c  */
                  /* has to stop rendering text when it receives an IDCMP_SIZEVERIFY message.  */
                  /*                                                                          */
                  /* if this example had instead asked to hear about IDCMP events that could  */
                  /* take place between SIZEVERIFY and NEWSIZE events (especially INTUITICKS), */
                  /* it should turn off those events here using ModifyIDCMP().                */
                  /*                                                                          */
                  /* After we allow the user to resize the window, we cannot write into the   */
                  /* window until the user has finished resizing it because we need the       */
                  /* window's new size to adjust the clipping area.  Specifically, we have    */
                  /* to wait for an IDCMP_NEWSIZE message which Intuition will send when the  */
                  /* user lets go of the resize gadget.  For now, we set the waitfornewsize   */
                  /* flag to stop rendering until we get that NEWSIZE message.                */

                   waitfornewsize = TRUE;
                   WaitBlit();

                   ReplyMsg((struct Message *)mymsg);        /* The blitter is done, let the */
               }                                             /* user resize the window.      */
```

```
            else
            {
                ReplyMsg((struct Message *)mymsg);
                waitfornewsize = FALSE;
                       /* The user has resized the window, so get the new window dimensions */
                myrectangle.MinX = mywin->BorderLeft;      /* and readjust the layers      */
                myrectangle.MinY = mywin->BorderTop;       /* clipping region accordingly.  */
                myrectangle.MaxX = mywin->Width - (mywin->BorderRight + 1);
                myrectangle.MaxY = mywin->Height - (mywin->BorderBottom + 1);
                InstallClipRegion(mywin->WLayer, NULL);
                ClearRegion(new_region);
                if (OrRectRegion(new_region, &myrectangle))
                    InstallClipRegion(mywin->WLayer, new_region);
                else
                {
                    aok = FALSE;
                    position = actual + 1;
                }
            }
        }
        if (mytask->tc_SigRecvd & SIGBREAKF_CTRL_C)             /* Check for user break.   */
        {
            aok = FALSE;
            position = actual + 1;
        }

        if (myrp->cp_y > (mywin->Height - (mywin->BorderBottom + 2))) /* if we reached the  */
        {                                                /* bottom of the page, clear the    */
            Delay(25);                                   /* rastport and move back to the top*/

            SetRast(myrp, 0);       /* Set the entire rastport to color zero.  This will not */
            Move(myrp,              /* the window borders because of the layers clipping.    */
            mywin->BorderLeft + 1,
            mywin->BorderTop + myfont->tf_YSize + 1);
         }
      }
   }
   if (actual < 0) VPrintf("Error while reading\n", NULL);
}
```

What Fonts Are Available?

The diskfont.library function **AvailFonts()** fills in a memory area designated by you with a list of all of the fonts available to the system. **AvailFonts()** searches the AmigaDOS directory path currently assigned to FONTS: and locates all available fonts. If you haven't issued a DOS Assign command to change the FONTS: directory path, it defaults to the sys:fonts directory.

```
LONG AvailFonts( struct AvailFontsHeader *mybuffer, LONG bufBytes, LONG flags );
```

AvailFonts() fills in a memory area, mybuffer, which is bufBytes bytes long, with an **AvailFontsHeader** structure:

```
struct AvailFontsHeader {
    UWORD   afh_NumEntries;      /* number of AvailFonts elements */
    /* struct AvailFonts afh_AF[], or struct TAvailFonts afh_TAF[]; */
};
```

This structure is followed by an array of **AvailFonts** structures with the number of entries in the array equal to **afh_NumEntries**:

```
struct AvailFonts {
    UWORD   af_Type;            /* MEMORY, DISK, or SCALED */
    struct TextAttr af_Attr;    /* text attributes for font */
};
```

Each **AvailFonts** structure describes a font available to the OS. The flags field lets **AvailFonts()** know which fonts you want to hear about. At present, there are four possible flags:

AFF_MEMORY — Create **AvailFonts** structures for all **TextFont**'s currently in the system list.

AFF_DISK — Create **AvailFonts** structures for all **TextFont**'s that are currently available from disk.

AFF_SCALED — Create **AvailFonts** structures for **TextFont**'s that do not have their FPF_DESIGNED flag set. If the AFF_SCALED flag is not present, **AvailFonts()** will not create **AvailFonts** structures for fonts that have been scaled, which do not have the FPF_DESIGNED flag set.

AFF_TAGGED — These **AvailFonts** structures are really **TAvailFonts** structures. These structures were created for Release 2 to allow **AvailFonts()** to list tag values:

```
struct TAvailFonts {
    UWORD   taf_Type;           /* MEMORY, DISK, or SCALED */
    struct TTextAttr taf_Attr;  /* text attributes for font */
};
```

Notice that AFF_MEMORY and AFF_DISK are not mutually exclusive; a font that is currently in memory may also be available for loading from disk. In this case, the font will appear twice in the array of **AvailFonts** (or **TAvailFonts**) structures.

If **AvailFonts()** fails without any major system problems, it will be because the buffer for the **AvailFontsHeader** structure was not big enough to contain all of the **AvailFonts** or **TAvailFonts** structures. In this case, **AvailFonts()** returns the number of additional bytes that mybuffer needed to contain all of the **TAvailFonts** or **AvailFonts** structures. You can then use that return value to figure out how big the buffer needs to be, allocate that memory, and try **AvailFonts()** again:

```
int afShortage, afSize;
struct AvailFontsHeader *afh;
 . . .
    afSize = AvailFonts(afh, 0L, AFF_MEMORY|AFF_DISK|AFF_SCALED|AFF_TAGGED);
    do
    {
        afh = (struct AvailFontsHeader *) AllocMem(afSize, 0);
        if (afh)
        {
            afShortage = AvailFonts(afh, afSize, AFF_MEMORY|AFF_DISK|AFF_SCALED|AFF_TAGGED);
            if (afShortage)
            {
                FreeMem(afh, afSize);
                afSize += afShortage;
            }
        }
        else
        {
            fail("AllocMem of AvailFonts buffer afh failed\n");
            break;
        }
    } while (afShortage); /* if (afh) non-zero here, then:                  */
                          /* 1. it points to a valid AvailFontsHeader,      */
                          /* 2. it must have FreeMem(afh, afSize) called for it after use. */
```

The following code, AvailFonts.c, uses **AvailFonts()** to find out what fonts are available to the system. It uses this information to open every available font (one at a time), print some information about the font (including the TA_DeviceDPI tag values if they are present), and renders a sample of the font into a clipping region.

```
;/* AvailFonts.c - Execute me to compile me with Lattice 5.10a
LC -cfistq -v -y -j73 AvailFonts.c
Blink FROM LIB:c.o,AvailFonts.o TO AvailFonts LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit ;*/

#include <exec/types.h>
#include <dos/rdargs.h>
#include <dos/dosextens.h>
#include <intuition/intuition.h>
#include <intuition/screens.h>
#include <graphics/text.h>
#include <graphics/displayinfo.h>
#include <graphics/regions.h>
#include <graphics/gfx.h>
#include <libraries/diskfont.h>
#include <utility/tagitem.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/layers_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/diskfont_protos.h>
#include <clib/utility_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\0$VER: AvailFonts 36.3";

void MainLoop(void);
ULONG StrLen(UBYTE *);

struct stringstruct {
        UBYTE *string;
        LONG charcount;
        WORD stringwidth;
};

UBYTE *alphabetstring = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
struct stringstruct fname, fheight, XDPI, YDPI, entrynum;
struct Library *DiskfontBase, *IntuitionBase, *LayersBase, *GfxBase, *UtilityBase;
struct Window *mywin;
struct RastPort *mycliprp, myrp;
struct Rectangle myrect;
struct Region *new_region, *old_region;
struct DrawInfo *mydrawinfo;
struct AvailFontsHeader *afh;
LONG fontheight, alphabetcharcount;
WORD stringwidth;

void main(int argc, char **argv)
{
  struct TextFont *defaultfont = NULL;
  struct TextAttr defaultfontattr = { "topaz.font", 9, 0, 0 };

  LONG afsize, afshortage, cliprectside;

  fname.string = "Font Name:  ";
  fheight.string = "Font Height:  ";
  XDPI.string = "X DPI:  ";
  YDPI.string = "Y DPI:  ";
  entrynum.string = "Entry #:  ";

  if (DiskfontBase = OpenLibrary("diskfont.library", 37L))             /* Open the libraries. */
  {
    if (IntuitionBase = OpenLibrary("intuition.library", 37L))
    {
      if (GfxBase = OpenLibrary("graphics.library", 37L))
      {
        if (LayersBase = OpenLibrary("layers.library", 37L))
        {
          if (UtilityBase = OpenLibrary("utility.library", 37L))
          {
```

```
if (mywin = OpenWindowTags(NULL,                              /* Open that window. */
                    WA_SmartRefresh,TRUE,
                    WA_SizeGadget,  FALSE,
                    WA_CloseGadget, TRUE,
                    WA_IDCMP,       IDCMP_CLOSEWINDOW,
                    WA_DragBar,     TRUE,
                    WA_DepthGadget, TRUE,
                    WA_Title,       (ULONG)"AvailFonts() example",
                    TAG_END))
{
  myrp = *(mywin->RPort);       /* A structure assign: clone my window's Rastport. */
                                /* RastPort.  This RastPort will be used to render */
                                /* the font specs, not the actual font sample.     */
  if (mydrawinfo = GetScreenDrawInfo(mywin->WScreen))
  {
    SetFont(&myrp, mydrawinfo->dri_Font);

    myrect.MinX = mywin->BorderLeft;                          /* LAYOUT THE WINDOW */
    myrect.MinY = mywin->BorderTop;
    myrect.MaxX = mywin->Width - (mywin->BorderRight + 1);
    myrect.MaxY = mywin->Height - (mywin->BorderBottom + 1);

    cliprectside = (myrect.MaxX - myrect.MinX) / 20;

    fname.charcount     = StrLen(fname.string);
    fheight.charcount   = StrLen(fheight.string);
    XDPI.charcount      = StrLen(XDPI.string);
    YDPI.charcount      = StrLen(YDPI.string);
    entrynum.charcount  = StrLen(entrynum.string);
    alphabetcharcount   = StrLen(alphabetstring);

    fontheight = (myrp.Font->tf_YSize) + 2;

    if (fontheight > ((myrect.MaxY - myrect.MinY) / 6))  /* If the default screen  */
    {                                                    /* font is more than one- */
      defaultfont = OpenFont(&defaultfontattr);          /* sixth the size of the  */
      SetFont(&myrp, defaultfont);                       /* window, use topaz-9.   */
      fontheight = (myrp.Font->tf_YSize) + 2;
    }

    fname.stringwidth   = TextLength(&myrp, (STRPTR)fname.string, fname.charcount);
    fheight.stringwidth = TextLength(&myrp, (STRPTR)fheight.string, fheight.charcount);
    XDPI.stringwidth    = TextLength(&myrp, (STRPTR)XDPI.string, XDPI.charcount);
    YDPI.stringwidth    = TextLength(&myrp, (STRPTR)YDPI.string, YDPI.charcount);
    entrynum.stringwidth =
        TextLength(&myrp, (STRPTR)entrynum.string, entrynum.charcount);

    stringwidth = fname.stringwidth;         /* What is the largest string length? */
    stringwidth =
        (fheight.stringwidth > stringwidth) ? fheight.stringwidth : stringwidth;
    stringwidth = (XDPI.stringwidth > stringwidth) ? XDPI.stringwidth : stringwidth;
    stringwidth = (YDPI.stringwidth > stringwidth) ? YDPI.stringwidth : stringwidth;
    stringwidth =
        (entrynum.stringwidth > stringwidth) ? entrynum.stringwidth : stringwidth;
    stringwidth += mywin->BorderLeft;

    if (stringwidth < ((myrect.MaxX - myrect.MinX) >> 1)) /* If the stringwidth is */
    {                                                  /* more than half the viewing*/
      SetAPen(&myrp, mydrawinfo->dri_Pens[TEXTPEN]);  /* area, quit because the    */
      SetDrMd(&myrp, JAM2);                            /* font is just too big.     */

      Move(&myrp, myrect.MinX + 8 + stringwidth - fname.stringwidth,
                   myrect.MinY + 4 + (myrp.Font->tf_Baseline));
      Text(&myrp, fname.string, fname.charcount);

      Move(&myrp, myrect.MinX + 8 + stringwidth - fheight.stringwidth,
                  myrp.cp_y + fontheight);
      Text(&myrp, fheight.string, fheight.charcount);

      Move(&myrp, myrect.MinX + 8 + stringwidth - XDPI.stringwidth,
                  myrp.cp_y + fontheight);
      Text(&myrp, XDPI.string, XDPI.charcount);

      Move(&myrp, myrect.MinX + 8 + stringwidth - YDPI.stringwidth,
                  myrp.cp_y + fontheight);
      Text(&myrp, YDPI.string, YDPI.charcount);
```

```
            Move(&myrp, myrect.MinX + 8 + stringwidth - entrynum.stringwidth,
                        myrp.cp_y + fontheight);
            Text(&myrp, entrynum.string, entrynum.charcount);

            myrect.MinX = myrect.MinX + cliprectside;
            myrect.MaxX = myrect.MaxX - cliprectside;
            myrect.MinY = myrect.MinY + (5 * fontheight) + 8;
            myrect.MaxY = myrect.MaxY - 8;

            SetAPen(&myrp, mydrawinfo->dri_Pens[SHINEPEN]);          /* Draw a box around */
            Move(&myrp, myrect.MinX - 1, myrect.MaxY + 1);           /* the cliprect.     */
            Draw(&myrp, myrect.MaxX + 1, myrect.MaxY + 1);
            Draw(&myrp, myrect.MaxX + 1, myrect.MinY - 1);

            SetAPen(&myrp, mydrawinfo->dri_Pens[SHADOWPEN]);
            Draw(&myrp, myrect.MinX - 1, myrect.MinY - 1);
            Draw(&myrp, myrect.MinX - 1, myrect.MaxY);

            SetAPen(&myrp, mydrawinfo->dri_Pens[TEXTPEN]);
                                /* Fill up a buffer with a list of the available fonts */
            afsize = AvailFonts((STRPTR)afh, 0L, AFF_MEMORY|AFF_DISK|AFF_SCALED|AFF_TAGGED);
            do
            {
              afh = (struct AvailFontsHeader *) AllocMem(afsize, 0);
              if (afh)
              {
                afshortage = AvailFonts((STRPTR)afh, afsize,
                                        AFF_MEMORY|AFF_DISK|AFF_SCALED|AFF_TAGGED);
                if (afshortage)
                {
                  FreeMem(afh, afsize);
                  afsize += afshortage;
                  afh = (struct AvailFontsHeader *)(-1L);
                }
              }
            } while (afshortage && afh);

            if (afh)
            {
              /* This is for the layers.library clipping region that gets attached to */
              /* the window.  This prevents the application from unnecessarily        */
              /* rendering beyond the bounds of the inner part of the window. For     */
              /* more information on clipping, see the Layers chapter of this manual. */

              if (new_region = NewRegion())                 /* More layers stuff */
              {
                if (OrRectRegion(new_region, &myrect));/* Even more layers stuff */
                {
                  /* Obtain a pointer to the window's rastport and set up some of    */
                  /* the rastport attributes.  This example obtains the text pen     */
                  /* for the window's screen using the GetScreenDrawInfo() function. */
                  mycliprp = mywin->RPort;
                  SetAPen(mycliprp, mydrawinfo->dri_Pens[TEXTPEN]);

                  MainLoop();
                }
                DisposeRegion(new_region);
              }
              FreeMem(afh, afsize);
            }
          }
          FreeScreenDrawInfo(mywin->WScreen, mydrawinfo);
        }
        CloseWindow(mywin);
      }
      CloseLibrary(UtilityBase);
    }
    CloseLibrary(LayersBase);
  }
  CloseLibrary(GfxBase);
 }
 CloseLibrary(IntuitionBase);
 }
 CloseLibrary(DiskfontBase);
}
}
```

```
void MainLoop(void)
{
  UWORD x;
  struct Task *mytask;
  struct IntuiMessage *mymsg;
  BOOL aok = TRUE;
  struct TAvailFonts *afont;
  struct TextFont *myfont;
  UBYTE buf[8];
  ULONG dpi;

  mytask = FindTask(NULL);
  afont = (struct TAvailFonts *)&(afh[1]);

  for (x = 0; (x < afh->afh_NumEntries); x++)
  {
    if (aok)
    {
      if (myfont = OpenDiskFont(&(afont->taf_Attr)))
      {
        SetAPen(&myrp, mydrawinfo->dri_Pens[BACKGROUNDPEN]);   /* Print the TextFont attributes. */
        RectFill( &myrp, stringwidth, mywin->BorderTop + 4,
                  mywin->Width - (mywin->BorderRight + 1), myrect.MinY - 2 );

        SetAPen(&myrp, mydrawinfo->dri_Pens[TEXTPEN]);
        Move( &myrp, stringwidth + mywin->BorderLeft,
              mywin->BorderTop + 4 + (myrp.Font->tf_Baseline) );
        Text( &myrp, (UBYTE *)myfont->tf_Message.mn_Node.ln_Name,
              StrLen((UBYTE *)myfont->tf_Message.mn_Node.ln_Name) );

        Move(&myrp, stringwidth + mywin->BorderLeft, myrp.cp_y + fontheight); /* Print the      */
        sprintf(buf, "%d\0", myfont->tf_YSize);                              /* font's Y Size. */
        Text(&myrp, buf, StrLen(buf));

        Move(&myrp, stringwidth + mywin->BorderLeft, myrp.cp_y + fontheight); /* Print the X DPI */
        dpi = GetTagData( TA_DeviceDPI, 0L,
                          ((struct TextFontExtension *)(myfont->tf_Extension))->tfe_Tags);
        if (dpi)
        {
            sprintf(buf, "%d\0", ((dpi & 0xFFFF0000)>>16));
            Text(&myrp, buf, StrLen(buf));
        }
        else Text(&myrp, "nil", 3L);

        Move(&myrp, stringwidth + mywin->BorderLeft, myrp.cp_y + fontheight); /* Print the Y DPI */
        if (dpi)
        {
            sprintf(buf, "%d\0", (dpi & 0x0000FFFF));
            Text(&myrp, buf, StrLen(buf));
        }
        else Text(&myrp, "nil", 3L);

        Move(&myrp, stringwidth + mywin->BorderLeft, myrp.cp_y + fontheight);     /* Print the */
        sprintf(buf, "%d\0", x);                                                   /* entrynum. */
        Text(&myrp, buf, StrLen(buf));

        SetFont(mycliprp, myfont);
        old_region = InstallClipRegion(mywin->WLayer, new_region); /* Install clipping rectangle */

        SetRast(mycliprp, mydrawinfo->dri_Pens[BACKGROUNDPEN]);
        Move( mycliprp, myrect.MinX, myrect.MaxY - (myfont->tf_YSize - myfont->tf_Baseline) );
        Text(mycliprp, alphabetstring, alphabetcharcount);

        Delay(100);

        new_region = InstallClipRegion(mywin->WLayer, old_region);  /* Remove clipping rectangle */

        while (mymsg = (struct IntuiMessage *)GetMsg(mywin->UserPort))
        {
            aok = FALSE;
            x = afh->afh_NumEntries;
            ReplyMsg((struct Message *)mymsg);
        }
```

```
        if (mytask->tc_SigRecvd & SIGBREAKF_CTRL_C)       /* Did the user hit CTRL-C (the shell */
        {                                                  /* window has to receive the CTRL-C)? */
            aok = FALSE;
            x = afh->afh_NumEntries;
            VPrintf("Ctrl-C Break\n", NULL);
        }
        CloseFont(myfont);
      }
    }
    afont++;
  }
}


ULONG StrLen(UBYTE *string)
{
    ULONG x = 0L;

    while (string[x++]);
    return(--x);
}
```

How is an Amiga Font Structured in Memory?

So far, this chapter has concentrated on using library functions to render text, letting the system worry about the layout of the underlying font data. As far as the OS representation of a loaded font is concerned, outline fonts and normal bitmap fonts are structured identically. Color fonts have some extras information associated with them and are discussed a little later. Every loaded font, including color fonts, has a **TextFont** structure associated with them:

```
struct TextFont {
    struct Message tf_Message;  /* reply message for font removal */
    UWORD   tf_YSize;
    UBYTE   tf_Style;
    UBYTE   tf_Flags;
    UWORD   tf_XSize;
    UWORD   tf_Baseline;
    UWORD   tf_BoldSmear;       /* smear to affect a bold enhancement */

    UWORD   tf_Accessors;
    UBYTE   tf_LoChar;
    UBYTE   tf_HiChar;
    APTR    tf_CharData;        /* the bit character data */

    UWORD   tf_Modulo;          /* the row modulo for the strike font data   */
    APTR    tf_CharLoc;         /* ptr to location data for the strike font  */
                                /*   2 words: bit offset then size           */
    APTR    tf_CharSpace;       /* ptr to words of proportional spacing data */
    APTR    tf_CharKern;        /* ptr to words of kerning data              */
};
```

The first field in this structure is a **Message** structure. The node in this **Message** structure is what the OS uses to link together the fonts in the system list. From this node, an application can extract a font's name. The other fields in the **TextFont** structure are as follows:

tf_YSize
: The maximum height of this font in pixels.

tf_Style
: The style bits for this particular font, which are defined in *<graphics/text.h>*. These include the same style bits that were mentioned in the discussion of the **TextAttr** structure in the "Choosing the Font" section of this chapter. In addition to those bits, there is also the FSF_COLORFONT bit, which identifies this as a special kind of **TextFont** structure called a **ColorTextFont** structure. This is discussed later in this chapter.

tf_Flags

The flags for this font, which were mentioned along with the style bits in the section, ''Choosing the Font''.

tf_XSize

If this font is monospaced (non-proportional), **tf_XSize** is the width of each character. The width includes the extra space needed to the left and right of the character to keep the characters from running together.

tf_Baseline

The distance in pixels from the top line of the font to the baseline.

tf_BoldSmear

When algorithmically bolding, the Amiga currently ''smears'' a glyph by rendering it, moving over **tf_BoldSmear** number of pixels, and rerendering the glyph.

tf_Accessors

The number of currently open instances of this font (like the open count for libraries).

tf_LoChar

This is the first character glyph (the graphical symbol associated with this font) defined in this font. All characters that have ASCII values below this value do not have an associated glyph.

tf_HiChar

This is the last character glyph defined in this font. All characters that have ASCII values above this value do not have an associated glyph. An application can use these values to avoid rendering characters which have no associated glyphs. Any characters without an associated glyph will have the default glyph associated to them. Normally, the default glyph is a rectangle.

tf_CharData

This is the address of the bitmap from which the OS extracts the font's glyphs. The individual glyphs are bit-packed together. The individual bitmaps of the glyphs are placed in ASCII order side by side, left to right. The last glyph is the default glyph. The following is what the bitmap of the suits-8 font example looks like (suits-8 is the complete, disk-based bitmap font example used later in this chapter):

```
.@@@...@@@......@........@.......@@@....@@@@@@@@@@@@............
@@@@@.@@@@@...@@@@@.....@@@.....@@@@@...@@........@@............
.@@@@@@@@@..@@@@@@@@@..@@@@@..@@..@..@@.@@........@@............
..@@@@@@@..@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@........@@............
...@@@@@....@@@.@.@@@..@@@@@..@@..@..@@.@@........@@............
....@@@.........@.......@@@.......@.....@@........@@............
.....@........@@@@@......@......@@@@@...@@@@@@@@@@@@............
................................................................
```

This font is rather sparse, as it only has five glyphs. Most fonts at least have glyphs for each letter of the alphabet. In this example, each glyph represents a symbol for a suit in a standard deck of cards (from left to right: hearts, spades, diamonds, and clubs). Notice that there is no space between the individual glyphs. The spacing information is kept in separate tables to reduce the amount of memory occupied by the font.

tf_Modulo

This is number of bytes the pointer must move to go from one line of a glyph to the next. This is the pixel width of the entire font bitmap divided by eight. Notice that the bitmap above does not stop after it gets to the end of the last glyph. It is padded with zero bits to the nearest WORD boundary.

tf_CharLoc

This is a pointer to the **CharLoc**, the character location table. This table tells the OS how far into the bitmap to look for a character and how many pixels to fetch from each row. The **CharLoc** table for the suits-8 font looks like this:

```
$0000000B,$000B000B,$00160007,$001D000B,$0028000C
```

Each of the five long words in this character location table corresponds to a glyph in Suits-8. Each long word is broken up into two word values. The first word is the offset in pixels from the left edge of the bitmap to the first column containing the corresponding glyph. The second word is the width in pixels of the corresponding glyph image in the bitmap (note, this is not the width of the actual glyph as the actual glyph will have some space on either side of it). For example, the diamond character (the third character) starts at offset $16 (22) and it is 7 pixels wide.

tf_CharSpace

This is a pointer to an array of WORDs containing the width of each glyph in the font. Each entry tells the OS how much to increment the current horizontal position (usually **RastPort.cp_X**). For reverse path fonts, these values can be negative.

tf_CharKern

This is a pointer to an array of "kerning" values. As it is used here, the term "kerning" is unorthodox. On the Amiga, kerning refers to the number pixels to leave blank before rendering a glyph. The normal typographical definition of the word refers to the number of pixels to back up before rendering the current glyph and is usually associated with a specific pair of glyphs rather than one particular glyph.

For each glyph the system renders, it has to do several things:

1) Get the value from the kerning table that corresponds to this glyph and begin the rendering that number of pixels to the right.

2) Find this glyph's bitmap using the **CharLoc** table and blit the glyph to the rastport.

3) If this is a proportional font, look in the spacing table and figure how many pixels to advance the rastport's horizontal position. For a monospaced font, the horizontal position advance comes from the **TextFont**'s **tf_XSize** field.

Under Release 2, when the system opens a new font, it creates an extension to the **TextFont** structure called the **TextFontExtension**. This extension is important because it contains a pointer to the font's tag list, which is where the system keeps the font's TA_DeviceDPI values. The **TextFont**'s **tf_Message.mn_ReplyPort** field contains a pointer to the **TextFontExtension** structure (the *<graphics/text.h>* include file #defines **tf_Message.mn_ReplyPort** as **tf_Extension**). The only field of interest in the **TextFontExtension** structure is:

```
struct TagItem *tfe_Tags;          /* Text Tags for the font */
```

which points to the font's tag list. Before accessing the tag list, an application should make sure that this font has a corresponding **TextFontExtension**. The **ExtendFont()** function will return a value of TRUE if this font already has an extension or **ExtendFont()** was able to create an extension for this font.

But What About Color Fonts?

When the Amiga loads a color font, it has to account for more information than will fit into the **TextFont** structures. For color fonts, the Amiga uses a superset of the **TextFont** structure called the **ColorTextFont** structure (defined in <*graphics/text.h*>):

```
struct ColorTextFont {
    struct TextFont ctf_TF;
    UWORD   ctf_Flags;      /* extended flags */
    UBYTE   ctf_Depth;      /* number of bit planes */
    UBYTE   ctf_FgColor;    /* color that is remapped to FgPen */
    UBYTE   ctf_Low;        /* lowest color represented here */
    UBYTE   ctf_High;       /* highest color represented here */
    UBYTE   ctf_PlanePick;      /* PlanePick ala Images */
    UBYTE   ctf_PlaneOnOff;     /* PlaneOnOff ala Images */
    struct ColorFontColors *ctf_ColorFontColors; /* colors for font */
    APTR    ctf_CharData[8];    /*pointers to bit planes ala tf_CharData */
};
```

The **ctf_TF** field is the **TextFont** structure described in the previous section. There are two minor differences between the data stored in a color font's **TextFont** structure and an ordinary **TextFont** structure. The first is that the color font's **TextFont.tf_Style** field has the FSF_COLORFONT bit set. The other difference is that the bitmap that **TextFont.tf_CharData** points to can be a multi-plane bitmap.

The **ctf_Flags** field is a bitfield that supports the following flags:

CT_COLORFONT	The color map for this font contains colors specified by the designer.
CT_GREYFONT	The colors for this font describe evenly stepped gray shades from low to high.

The **ctf_Depth** field contains the bitplane depth of this font's bitmap.

The **ctf_FgColor** contains the color that will be dynamically remapped during output by changing **ctf_FgColor** to **RastPort.FgPen**. This field allows a **ColorTextFont** to contain color outlines, shadows, etc. while also containing a predominant color that can be changed by the user. If the font does not have a predominant color, **ctf_FgColor** is 0xFF. For example, given a color font that has a blue and red outline and a white center, the person designing the font can set **ctf_FgColor** equal to white. Then when the font is used in a paint package that supports color fonts, the white will change to the current foreground color.

The fields **ctf_Low** and **ctf_High** contain the lowest and highest color values in the **ColorTextFont**. For example, a four bitplane color font can have sixteen colors, but the font may use only nine of those colors, thus **ctf_Low**=0 and **ctf_High**=8. The most important use of these colors is for defining the boundaries of a gray scale font. If the font uses less than the total number of colors around but needs white as the lowest and black as the highest level of gray, the boundaries would have to be defined in order for the font to be rendered correctly. Defaults for these values should be the lowest and the highest values for the given number of bitplanes.

The **ctf_PlanePick** and **ctf_PlaneOnOff** contain information for saving space in memory for some types of **ColorTextFont** structures. The **ctf_PlanePick** field contains information about where each plane of data will be rendered in a given bitmap. The **ctf_PlaneOnOff** field contains information about planes that are not used to render a plane of font data. If **ctf_PlaneOnOff** contains a zero bit for a given plane, that bitplane is cleared. If **ctf_PlaneOnOff** contains a set bit for a given plane, that bitplane is filled. For more information on how the **ctf_PlaneOnOff** and **ctf_PlanePick** fields work see the ''Specifying the Colors of a Bob'' section of the ''Graphics Sprites, Bobs and Animation'' chapter of this book.

The **ctf_ColorFontColors** field contains a pointer to a **ColorFontColors** structure:

```
struct ColorFontColors {
    UWORD   cfc_Reserved;       /* *must* be zero */
    UWORD   cfc_Count;          /* number of entries in cfc_ColorTable */
    UWORD   *cfc_ColorTable;    /* 4 bit per component color map packed xRGB */
};
```

Which specifies the colors used by this font. The **ColorFontColors cfc_Count** field contains the number of colors defined in this structure. Each color is defined as a single, UWORD entry in the **cfc_ColorTable**. For each entry in **cfc_ColorTable**, the lowest four bits make up the blue element, the next four bits the green element, the next four bits the red element, and the upper four bits should be masked out.

The **ctf_CharData[]** fields is an array of pointers to each of the bitplanes of the color font data.

The Composition of a Bitmap Font on Disk

For each Amiga bitmap font stored on disk (normally in the FONTS: assign directory), there is a corresponding ".font" file, a directory, and within that directory, a series of files bearing numeric names. For example, for the font Sapphire, within FONTS:, there is a file called sapphire.font, a directory called Sapphire, and within the directory Sapphire are the files 14 and 19.

For a bitmap font (including color fonts), the ".font" file is a **FontContentsHeader** structure:

```
struct FontContentsHeader {
    UWORD   fch_FileID;     /* FCH_ID */
    UWORD   fch_NumEntries;     /* the number of FontContents elements */
    struct FontContents fch_FC[]; /* or struct TFontContents fch_TFC[]; */
};

#define MAXFONTPATH 256
```

Where the **fch_FileID** field can be be either:

FCH_ID	0x0f00	This **FontContentsHeader** uses **FontContents** structures to describe the available sizes of this font.
TFCH_ID	0x0f02	This **FontContentsHeader** uses **TFontContents** structures to describe the available sizes of this font.

The **fch_FileID** can also be equal to 0x0f03, but that is only for scalable outline fonts.

The **FontContents** structure:

```
struct FontContents {
    char    fc_FileName[MAXFONTPATH];
    UWORD   fc_YSize;
    UBYTE   fc_Style;
    UBYTE   fc_Flags;
};
```

describes one of the sizes of this font that is available to the system as a designed font size. For each **FontContents** structure, there should be a corresponding font descriptor file in this font's directory that contains data for this size font. The FontContents fields correspond to the similarly named field in the **TextFont** structure.

The **TFontContents** structure is almost the same as the **FontContents** structure except that it allows the OS to store tag value pairs in the extra space not used by the file name. Currently, this allows the OS to preserve the X and Y DPI (TA_DeviceDPI) values for a font.

```
struct TFontContents {
    char     tfc_FileName[MAXFONTPATH-2];
    UWORD    tfc_TagCount;       /* including the TAG_DONE tag */
    /*
     *  if tfc_TagCount is non-zero, tfc_FileName is overlaid with
     *  Text Tags starting at:  (struct TagItem *)
     *      &tfc_FileName[MAXFONTPATH-(tfc_TagCount*sizeof(struct TagItem))]
     */
    UWORD    tfc_YSize;
    UBYTE    tfc_Style;
    UBYTE    tfc_Flags;
};
```

The **fch_NumEntries** contains the number of font sizes (and the number of **FontContents** or **TFontContents** structures) that this ".font" file describes. The **fch_FC[]** is the array of **FontContents** or **TFontContents** structures that describe this font.

For each font size described in a **FontContents** (or **TFontContents**) structure, there is a corresponding file in that font's directory whose name is its size. For example, for the font size Sapphire-19, there is a file in the Sapphire directory called 19. That file is basically a **DiskFontHeader** disguised as a loadable DOS hunk and is known as a font descriptor file. This allows the diskfont.library to use the dos.library to load the module just like it was a hunk of relocatable 680x0 instructions. It even contains two instructions before the real **DiskFontHeader** structure that will cause the 680x0 to stop running the **DiskFontHeader** if it does inadvertently get executed.

```
#define  MAXFONTNAME     32      /* font name including ".font " */

struct DiskFontHeader {
    /* the following 8 bytes are not actually considered a part of the  */
    /* DiskFontHeader, but immediately precede it. The NextSegment is    */
    /* supplied by the linker/loader, and the ReturnCode is the code     */
    /* at the beginning of the font in case someone runs it...           */
    /*   ULONG dfh_NextSegment;                     actually a BPTR      */
    /*   ULONG dfh_ReturnCode;                      MOVEQ #0,D0 : RTS    */
    /* here then is the official start of the DiskFontHeader...          */
    struct Node dfh_DF;            /* node to link disk fonts */
    UWORD    dfh_FileID;           /* DFH_ID */
    UWORD    dfh_Revision;         /* the font revision */
    LONG     dfh_Segment;          /* the segment address when loaded */
    char     dfh_Name[MAXFONTNAME]; /* the font name (null terminated) */
    struct TextFont dfh_TF;        /* loaded TextFont structure */
};

/* unfortunately, this needs to be explicitly typed */
/* used only if dfh_TF.tf_Style FSB_TAGGED bit is set */
#define dfh_TagList     dfh_Segment     /* destroyed during loading */
```

The **dfh_DF** field is an Exec **Node** structure that the diskfont library uses to link together the fonts it has loaded. The **dfh_FileID** field contains the file type, which currently is DFH_ID (defined in *<libraries/diskfont.h>*). The **dfh_Revision** field contains a revision number for this font. The **dfh_Segment** field will contain the segment address when the font is loaded. The **dfh_FontName** field will contain the font's name after the font descriptor is **LoadSeg()**'ed. The last field, **dfh_TextFont** is a **TextFont** structure (or **ColorTextFont** structure) as described in the previous section. The following is a complete example of a proportional, bitmap font.

```
* A sparse (but complete) sample font.
*
* 1.  Assemble this file (assumed to have been saved as "suits8.asm"). For example, if you have the
* CAPE 680x0 assembler, and you have assigned "include:" to the directory containing your include
* files, use:
*           CAsm -a "suits8.asm" -o "suits8.o" -i "include:"
*
*      Link "suits8.o".  For example, if you have Lattice, use:
*           BLink from "suits8.o" to "suits8"
*
* 2.  Create the subdirectory "Fonts:suits".  Copy the file "suits8" (created in step 1)
*     to "Fonts:suits/8".
*
* 3.  Create a font contents file for the font.  You can do this by two methods:
*
*   a. Run the program "System/FixFonts" which will create the file "Fonts:suits.font"
*      automatically.
*   b. Use the NewFontContents() call in the diskfont library to create a FontContentsHeader
*      structure, which can be saved in the Fonts: directory as "suits.font".  This is essentially
*      what FixFonts does.
*
* The next word contains the font YSize; in this case, 0x0008.
*
* The next byte contains the font Flags, in this case 0x00.
*
* The last byte contains the font characteristics, in this case 0x60.  This says it is a disk-based
* font (bit 1 set) and the font has been removed (bit 7 set), saying that the font is not currently
* resident.
*
* Summary of suits.font file:
*
* Name: fch_FileID fch_NumEntries fc_FileName     fc_YSize fc_Flags fc_Style
* Size: word       word        MAXFONTPATH bytes word     byte     byte
* Hex:  0f00       0001        73756974732F3800  0008     00       60
* ASCII:            s u i t s / 8 \0
*
* The correct length of a font file may be calculated with this formula:
* length = ((number of font contents entries) * (MAXFONTPATH+4)) + 4. In this case (one entry),
* this becomes (MAXFONTPATH + 8) or 264.
*
* To try out this example font, do the following.  Use the Fonts Preferences editor or a
* program that allows the user to select fonts.  Choose the "suits" font in size 8.
* This example font defines ASCII characters 'a' 'b' 'c' and 'd' only. All other characters
* map to a rectangle, meaning "character unknown".

    INCLUDE   "exec/types.i"
    INCLUDE   "exec/nodes.i"
    INCLUDE   "libraries/diskfont.i"

    MOVEQ     #-1,D0        ; Provide an easy exit in case this file is
    RTS                     ; "Run" instead of merely loaded.

    DC.L      0             ; ln_Succ     * These five entries comprise a Node structure,
    DC.L      0             ; ln_Pred     * used by the system to link disk fonts into a
    DC.B      NT_FONT       ; ln_Type     * list.  See the definition of the "DiskFontHeader"
    DC.B      0             ; ln_Pri      * structure in the "libraries/diskfont.i" include
    DC.L      fontName      ; ln_Name     * file for more information.

    DC.W      DFH_ID        ; FileID
    DC.W      1             ; Revision
    DC.L      0             ; Segment

* The next MAXFONTNAME bytes are a placeholder.  The name of the font contents file (e.g.
* "suits.font") will be copied here after this font descriptor is LoadSeg-ed into memory.  The Name
* field could have been left blank, but inserting the font name and size (or style) allows one to
* tell something about the font by using "Type OPT H" on the file.

fontName:
    DC.B      "suits8"      ; Name

* If your assembler needs an absolute value in place of the "length" variable, simply count the
* number of characters in Name and use that.

length  EQU     *-fontName              ; Assembler calculates Name's length.
        DCB.B   MAXFONTNAME-length,0    ; Padding of null characters.
```

```
font:
    DC.L    0                 ; ln_Succ       * The rest of the information is a TextFont
    DC.L    0                 ; ln_Pred       * structure.  See the "graphics/text.i" include
    DC.B    NT_FONT           ; ln_Type       * file for more information.
    DC.B    0                 ; ln_Pri
    DC.L    fontName          ; ln_Name
    DC.L    0                 ; mn_ReplyPort
    DC.W    0                 ; (Reserved for 1.4 system use.)
    DC.W    8                 ; tf_YSize
    DC.B    0                 ; tf_Style
    DC.B    FPF_DESIGNED!FPF_PROPORTIONAL!FPF_DISKFONT ; tf_Flags
    DC.W    14                ; tf_XSize
    DC.W    6                 ; tf_Baseline  <----* tf_Baseline must be no greater than
    DC.W    1                 ; tf_BoldSmear      * tf_YSize-1, otherwise algorithmically-
    DC.W    0                 ; tf_Accessors      * generated styles (italic in particular)
    DC.B    97                ; tf_LoChar         * can corrupt system memory.
    DC.B    100               ; tf_HiChar
    DC.L    fontData          ; tf_CharData
    DC.W    8                 ; tf_Modulo  <- * add this to the data pointer to go from from
                              ;               * one row of a character to the next row of it.
    DC.L    fontLoc           ; tf_CharLoc <---------------- * bit position in the font data
    DC.L    fontSpace         ; tf_CharSpace                 * at which the character begins.
    DC.L    fontKern          ; tf_CharKern

* The four characters of this font define the four playing-card suit symbols.  The heart, club,
* diamond, and spade map to the lower-case ASCII characters 'a', 'b', 'c', and 'd' respectively The
* fifth entry in the table is the character to be output when there is no entry defined in the
* character set for the requested ASCII value.  Font data is bit-packed edge to edge to save space.

fontData:                                ;   97 (a)       98 (b)       99 (c)   100 (d)       255
    DC.W  $071C0,$08040,$070FF,$0F000 ; <          X           X    X          X         >
    DC.W  $0FBE3,$0E0E0,$0F8C0,$03000 ;  .@@@...@@@.  .....@.....  ...@...  ....@@@....  @@@@@@@@@@@@
    DC.W  $07FCF,$0F9F3,$026C0,$03000 ;  @@@@@.@@@@@  ...@@@@@...  ..@@@..  ...@@@@@...  @@........@@
    DC.W  $03F9F,$0FFFF,$0FFC0,$03000 ;  .@@@@@@@@@.  .@@@@@@@@@.  .@@@@@.  .@@..@..@@.  @@........@@
    DC.W  $01F0E,$0B9F3,$026C0,$03000 ;  ..@@@@@@@..  @@@@@@@@@@@  @@@@@@@  @@@@@@@@@@@  @@........@@
    DC.W  $00E00,$080E0,$020C0,$03000 ;  ...@@@@@...  .@@@.@.@@@.  .@@@@@.  .@@..@..@@.  @@........@@
    DC.W  $00403,$0E040,$0F8FF,$0F000 ;  ....@@@....  .....@.....  ..@@@..  .....@.....  @@........@@
    DC.W  $00000,$00000,$00000,$00000 ;  .....@.....  ...@@@@@...  ...@...  ...@@@@@...  @@@@@@@@@@@@
    DC.W  $00000,$00000,$00000,$00000 ;  ...........  ...........  .......  ...........  ............

fontLoc:                  ; The fontLoc information is used to "unpack" the fontData. Each
    DC.L    $00000000B    ; pair of words specifies how the characters are bit-packed.  For
    DC.L    $0000B000B    ; example, the first character starts at bit position 0x0000, and
    DC.L    $000160007    ; is 0x000B (11) bits wide.  The second character starts at bit
    DC.L    $0001D000B    ; position 0x000B and is 0x000B bits wide, and so on.  This tells
    DC.L    $00028000C    ; the font handler how to unpack the bits from the array.

fontSpace:                         ; fontSpace array:  Use a space this wide to contain this
    DC.W    000012,000012          ; character when it is printed.  For reverse-path fonts
    DC.W    000008,000012,000013   ;  these values would be small or negative.

fontKern:                          ; fontKern array:  Place a space this wide after the
    DC.W    000001,000001          ; corresponding character to separate it from the following
    DC.W    000001,000001,000001   ; character.  For reverse-path fonts these values would be large
                                   ; negative numbers, approximately the width of the characters.
fontEnd:
    END
```

Function Reference

The following are brief descriptions of the Graphics and Diskfont library functions that deal with text. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 29-1: Graphics Library Text Functions

Function	Description
Text()	Render a text string to a **RastPort**.
SetFont()	Set a **RastPort**'s font.
AskFont()	Get the **TextAttr** for a **RastPort**'s font.
OpenFont()	Open a font currently in the system font list.
CloseFont()	Close a font.
AddFont()	Add a font to the system list.
RemFont()	Remove a font from the system list.
StripFont()	Remove the **tf_Extension** from a font (V36).
WeighTAMatch()	Get a measure of how well two fonts match (V36).
ClearScreen()	Clear **RastPort** from the current position to the end of the **RastPort**.
ClearEOL()	Clear **RastPort** from the current position to the end of the line.
AskSoftStyle()	Get the soft style bits of a **RastPort**'s font.
SetSoftStyle()	Set the soft style bits of a **RastPort**'s font.
TextLength()	Determine the horizontal raster length of a text string using the current **RastPort** settings.
TextExtent()	Determine the raster extent (along the X and Y axes) of a text string using the current **RastPort** settings (V36).
FontExtent()	Fill in a **TextExtent** structure with the bounding box for the characters in the specified font (V36).
TextFit()	Count the number of characters in a given string that will fit into a given bounds, using the current **RastPort** settings (V36).

Table 29-2: Diskfont Library Text Functions

Function	Description
AvailFonts()	Inquire which fonts are available from disk and/or memory.
NewFontContents()	Create a **FontContents** image for a font.
DisposeFontContents()	Free the result from **NewFontContents()**.
NewScaledDiskFont()	Create a **DiskFont** scaled from another font (V36).
OpenDiskFont()	Open a font, loading it from disk if necessary.

Chapter 30
LAYERS LIBRARY

This chapter describes the layers library which provides routines that are used to manage overlapping rectangular drawing areas that share a common display. Intuition uses the layers library to manage its system of windows.

The chapter also describes the use of *regions*, special structures used to mask off areas where drawing can take place. Regions are installed through the layers library function **InstallClipRegion()** but the routines for the creation, disposal and manipulation of regions are part of the graphics library.

Layers

The concept of a *layer* is closely tied to Intuition windows. A layer is a rectangular drawing area. A layer can overlap other layers and has a display priority that determines whether it will appear in front or behind other layers. Every Intuition window has an associated **Layer** structure. Layers allow Intuition and application programs to :

- Share a display's **BitMap** among various tasks in an orderly way by creating layers, separate drawing rectangles, within the **BitMap**.
- Move, size or depth-arrange a layer while automatically keeping track of which portions of other layers are hidden or revealed by the operation.
- Manage the remapping of coordinates, so the application need not track the layer's offset into the **BitMap**.
- Maintain each layer as a separate entity, which may optionally have its own **BitMap**.
- Automatically update same newly visible portions.

The layers library takes care of housekeeping: the low level, repetitive tasks which are required to keep track of where to place bits. The layers library also provides a locking mechanism which coordinates display updating when multiple tasks are drawing graphics to layers. The windowing environment provided by the Intuition library is largely based on layers.

WARNING: Layers may not be created or used directly with Intuition screens. Intuition windows are the only supported method of adding layers to Intuition screens. *Only* the layer locking and unlocking functions are safe to use with Intuition. An application must create and manage its own **View** if it will be creating layers directly on the display.

THE LAYER STRUCTURE

The internal representation of layers is essentially a set of clipping rectangles. Each layer is represented by an instance of the **Layer** structure. All the layers in a display are linked together through the **Layer_Info** structure. Any display shared by multiple layers (such as an Intuition screen) requires one **Layer_Info** data structure to handle interactions between the various layers. Here is a partial listing of the **Layer** structure from *<graphics/clip.h>*. (For a complete listing refer to the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*)

```
struct Layer
{
    struct  Layer *front,*back;
    struct  ClipRect    *ClipRect;  /* read by ROMs to find first cliprect */
    struct  RastPort    *rp;
    struct  Rectangle   bounds;
      ...

    UWORD   Flags;                  /* obscured ?, Virtual BitMap? */
    struct  BitMap *SuperBitMap;
      ...

    struct  Region  *DamageList;    /* list of rectangles to refresh through */
};
```

The Layer Structure is Read-Only. Applications should never directly modify any of the elements of the **Layer** structure. In addition, applications should only read the **front**, **back**, **rp**, **bounds**, **Flags**, **SuperBitMap** and **DamageList** elements of the **Layer** structure. (Some of these elements are subject to dynamic change by the system so proper layer locking procedures must be followed when relying on what the application has read.)

THE LAYER'S RASTPORT

When a layer is created, a **RastPort** is automatically to go along with it. The pointer to the **RastPort** is contained in the layer data structure. Using this **RastPort**, the application may draw anywhere into the layer's **bounds** rectangle. If the application tries to draw outside of this rectangle, the graphics routines will clip the graphics.

Here is sample code showing how to access the layer's **RastPort**:

```
struct RastPort *myRPort;    /* allocate a RastPort pointer for each layer */

myRPort = layer->rp;

/* The layer's RastPort may be used with any of the graphics library calls
** that require this structure.  For instance, to fill layer with color:
*/
SetRast(layer->rp, color);

/* set up for writing text into layer */
SetDrMd(layer->rp,JAM1);
SetAPen(layer->rp,0);
Move(layer->rp, 5, 7);

/* write into layer */
Text(layer->rp, string, strlen(string));
```

TYPES OF LAYERS

The layers library supports three types of layers: simple refresh, smart refresh and super bitmap. The type of the layer, specified by the **Flags** field in the **Layers** structure, determines what facilities the layer provides.

> *Use Only One Layer Type Flag.* The three layer-type **Flags** are mutually exclusive. That is, only one layer-type flag (LAYERSIMPLE, LAYERSMART and LAYERSUPER) should be specified.

Simple Refresh Layer

When an application draws into the layer, any portion of the layer that is visible (not obscured) will be rendered into the common **BitMap** of the viewing area. All graphics rendering routines are "clipped", so that only exposed sections of the layer are drawn into. No back-up of obscured areas is provided.

If another layer operation is performed that causes an obscured part of a simple refresh layer to be exposed, the application must determine if the section need be refreshed, re-drawing the newly exposed part of the layer as required.

The basic advantage of simple refresh is that it does not require back-up area to save drawing sections that cannot be seen, saving memory. However, the application needs to monitor the layer to see if it needs refreshing. This is typically performed with statements like:

```
if (layer->Flags & LAYERREFRESH)
    refresh(layer);
```

Smart Refresh Layer

Under smart refresh, the system provides dynamic backup of obscured sections of the layer. The graphics routines will automatically draw into these backup areas when they encounter an obscured part of the layer. The backup memory will be used to automatically update the display when obscured sections later become exposed.

With smart refresh layers, the system handles all of the refresh requirements of the layer, except when the layer is made larger. When parts of the layer are exposed by a sizing operation, the application must refresh the newly exposed areas.

The advantage of smart refresh is the speed of updating exposed regions of the layer and the ability of the system to manage part of the updating process for the application.. Its main disadvantage is the additional memory required to handle this automatic refresh.

Super Bitmap Layer

A super bitmap layer is similar to a smart refresh layer. It too has a back-up area for rendering graphics for currently obscured parts of the display. Whenever an obscured area is made visible, the corresponding part of the backup area is copied to the display automatically.

However, it differs from smart refresh in that:

- The back-up **BitMap** is user-supplied, rather than being allocated dynamically by the system.
- The back-up **BitMap** may be as large or larger than the the current size of the layer. It may also be larger than the maximum size of the layer.

To see a larger portion of a super bitmap on-display, use **SizeLayer()**. To see a different portion of the super bitmap in the layer, use **ScrollLayer()**.

When the graphics routines perform drawing commands, part of the drawing appears in the common **BitMap** (the on-display portion). Any drawing outside the displayed portion itself is rendered into the super bitmap. When the layer is scrolled or sized, the layer contents are copied into the super bitmap, the scroll or size positioning is modified, and the appropriate portions are then copied back into the layer. (Refer to the graphics library functions **SyncSBitMap()** and **CopySBitMap()**.

Backdrop Layer

A layer of any type may be designated a backdrop layer which will always appear behind all other layers. They may not be moved, sized, or depth-arranged. Non-backdrop layers will always remain in front of backdrop layers regardless of how the non-backdrop layer is moved, sized or depth-arranged.

OPENING THE LAYERS LIBRARY

Like all libraries, the layers library must be opened before it may be used. Check the Layers Autodocs to determine what version of the library is required for any particular Layers function.

```
struct Library *LayersBase;

if (NULL != (LayersBase = OpenLibrary("layers.library",33L)))
    {
    /* use Layers library */

    CloseLibrary((struct Library *)LayersBase);
    }
```

WORKING WITH EXISTING LAYERS

A common operation performed by applications is to render text or graphics into an existing layer such as an Intuition window. To prevent Intuition from changing the layer (for instance when the user resizes or moves the window) during a series of graphic operations, the layers library provides locking functions for obtaining exclusive access to a layer.

These locking functions are also useful for applications that create their own layers if the application has more than one task operating on the layers asynchronously. These calls coordinate multiple access to layers.

Table 30-1: Functions for Intertask Control of Layers (Layers Library)

LockLayer()	Lock out rendering in a single layer.
UnlockLayer()	Release **LockLayer()** lock.
LockLayers()	Lock out rendering in all layers of a display.
UnlockLayers()	Release **LockLayers()** lock.
LockLayerInfo()	Gain exclusive access to the display's layers.
UnlockLayerInfo()	Release **LockLayerInfo()** lock.

The following routines from the graphics library also allow multitasking access to layer structures:

Table 30-2: Functions for Intertask Control of Layers (Graphics Library)

LockLayerRom()	Same as **LockLayer()**, from layers library.
UnlockLayerRom()	Release **LockLayerRom()** lock.
AttemptLockLayerRom()	Lock layer only if it is immediately available.

These functions are similar to the layers **LockLayer()** and **UnlockLayer()** functions, but do not require the layers library to be open. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details.

Intertask Operations

If multiple tasks are manipulating layers on the same display they will be sharing a **Layer_Info** structure and their use of it and its related data structures need to be coordinated. To ensure that a structure remains cohesive, it should be operated on by only one task at a time. The **Layer_Info** encompasses all the layers existing on a single display.

LockLayerInfo() must be called whenever the visible portions of layers may be affected, or when the **Layer_Info** structure is changed.

```
void LockLayerInfo( struct Layer_Info *li );
```

The lock should be obtained whenever a layer is created, deleted sized or moved, as the list of layers that is being managed by the **Layer_Info** data structure must be updated.

It is not necessary to lock the **Layer_Info** data structure while rendering, or when calling routines like **ScrollLayer()**, because layer sizes and on-display positions are not being affected.

Use **UnlockLayerInfo()** when you have finished the layer operation:

```
void UnlockLayerInfo( struct Layer_Info *li );
```

If you don't unlock the **Layer_Info** then any other task calling **LockLayerInfo()** on the same **Layer_Info** structure will be blocked creating a potential deadlock situation.

In addition to locking the **Layer_Info** structure, the layer itself should be locked if it is shared between tasks so that only one task at a time renders graphics to it. **LockLayer()** is used to get exclusive graphics output to a layer.

```
void LockLayer( long dummy, struct Layer *layer );
```

If a graphics function is in process, the lock will return when the function is completed. Other tasks are blocked only if they attempt to draw graphics into this layer, or try to obtain a lock on this layer. The **MoveLayer()**, **SizeLayer()** and **ScrollLayer()** functions automatically lock and unlock the layer they operate on.

UnlockLayer() should be used after the graphics operation to make the layer available to other tasks again.

```
void UnlockLayer( struct Layer *layer );
```

If more than one layer must be locked, then the **LockLayer()** calls should be surrounded by **LockLayerInfo()** and **UnlockLayerInfo()** calls, to prevent deadlock situations.

The layers library provides two additional functions, **LockLayers()** and **UnlockLayers()**, for locking multiple layers.

```
void LockLayers( struct Layer_Info *li );
void UnlockLayers( struct Layer_Info *li );
```

LockLayers() is used to lock all layers in a single command. **UnlockLayers()** releases the layers lock. The system calls these routines during the **BehindLayer()**, **UpfrontLayer()** and **MoveLayerInFrontOf()** operations (described below).

Determining Layer Position

If the viewing area has been separated into several layers, the application may need to find out which layer is topmost at a particular x,y coordinate. Use the **WhichLayer()** function for this:

```
struct Layer *WhichLayer( struct Layer_Info *li, long x, long y );
```

To be sure that no task adds, deletes, or changes the sequence of layers before this information is used, call **LockLayerInfo()** before calling **WhichLayer()**, and call **UnlockLayerInfo()** when the operation is complete. In this way, the program may ensure that it is acting on valid information. Always check for a NULL return value (coordinate not in a layer) from **WhichLayer()**.

CREATING AND USING NEW LAYERS

The functions described in this section are generally not safe to use with Intuition. To create new layers for Intuition you use Intuition window calls (see the "Intuition Windows" chapter earlier in this book).

Only applications that create and mange their own **View** will be able to call the layer creation and updating functions discussed here.

Table 30-3: Functions for Creating and Updating Layers

NewLayerInfo()	**Allocating a Layer_Info structure.**
DisposeLayerInfo()	Deallocating a **Layer_Info** structure.
CreateUpfrontLayer()	Make a new layer in front of others.
CreateBehindLayer()	Make a new layer behind others.
DeleteLayer()	Remove and delete an existing layer.
MoveLayer()	Change the position (not depth) of a layer.
SizeLayer()	Change the size of a layer.
ScrollLayer()	Change the internal coordinates of a layer.
BehindLayer()	Depth arrange a layer behind others.
UpfrontLayer()	Depth arrange a layer in front of others.
MoveLayerInFrontOf()	Depth arrange a layer to a specific position.
SwapBitsRastPortClipRect()	Fast, non-layered and non-damaging display operation.
BeginUpdate()	Synchronize optimized refreshing for layer.
EndUpdate()	End optimized layer refresh.

Creating a Viewing Workspace

A viewing workspace may be created by using the primitives **InitVPort()**, **InitView()**, **MakeVPort()**, **MrgCop()**, and **LoadView()**. Please reference the "Graphics Primitives" chapter for details on creating a low-level graphics display. Do not create Layers directly on Intuition screens. Windows are the only supported method of creating a layer on a screen.

Creating the Layers

The application must first allocate and initialize a **Layer_Info** data structure which the system uses to keep track of layers that are created, use statements like:

```
struct Layer_Info *theLayerInfo;

if (NULL != (theLayerInfo = NewLayerInfo()))
    {
    /* use Layer_Info */

    DisposeLayerInfo(theLayerInfo);
    }
```

Layers may be created in the common bit map by calling **CreateUpfrontLayer()** or **CreateBehindLayer()**, with a sequence such as the following:

```
struct Layer      *layer;
struct Layer_Info *theLayerInfo;
struct BitMap     *theBitMap;

/* requests construction of a smart refresh layer. */
if (NULL == (layer = CreateUpfrontLayer(theLayerInfo, theBitMap,
          20, 20, 100, 80, LAYERSMART, NULL)))
    error("CreateUpfrontLayer() failed.");
else
    {
    ; /* layer successfully created here. */
    }
```

Allocating and Deallocating Layer_Info

Use **NewLayerInfo()** to allocate and initialize a **Layer_Info** structure and associated sub-structures.

```
struct Layer_Info *NewLayerInfo( void );
```

You *must* call this function before attempting to use any of the other layers functions described below. When you have finished with a **Layer_Info** structure, use **DisposeLayerInfo()** to deallocate it.

```
void DisposeLayerInfo( struct Layer_Info *li );
```

This function deallocates a **Layer_Info** and associated structures previously allocated with **NewLayerInfo()**.

Allocating and Deallocating Layers

Layers are created using the routines **CreateUpfrontLayer()** and **CreateBehindLayer()**. **CreateUpfrontLayer()** creates a layer that will appear in front of any existing layers.

```
struct Layer *CreateUpfrontLayer( struct Layer_Info *li, struct BitMap *bm,
    long x0, long y0, long x1, long y1, long flags, struct BitMap *bm2 );
```

CreateBehindLayer() creates a layer that appears behind existing layers, but in front of backdrop layers.

```
struct Layer *CreateBehindLayer( struct Layer_Info *li, struct BitMap *bm,
    long x0, long y0, long x1, long y1, long flags, struct BitMap *bm2 );
```

Both of these routines return a pointer to a **Layer** data structure (as defined in the include file <*graphics/layers.h*>), or NULL if the operation was unsuccessful.

> *A New Layer Also Gets a RastPort.* When a layer is created, the routine automatically creates a **RastPort** to go along with it. If the layer's **RastPort** is passed to the drawing routines, drawing will be restricted to the layer. See "The Layer's RastPort" section above.

Use the **DeleteLayer()** call to remove a layer:

```
LONG DeleteLayer( long dummy, struct Layer *layer );
```

DeleteLayer() removes a layer from the layer list and frees the memory allocated by the layer creation calls listed above.

Moving and Sizing Layers

The layers library includes three functions for moving and sizing layers:

```
LONG MoveLayer( long dummy, struct Layer *layer, long dx, long dy );
LONG SizeLayer( long dummy, struct Layer *layer, long dx, long dy );
LONG MoveSizeLayer( struct Layer *layer, long dx, long dy, long dw, long dh);
```

MoveLayer() moves a layer to a new position relative to its current position. **SizeLayer()** changes the size of a layer by modifying the coordinates of the lower right corner of the layer. **MoveSizeLayer()** changes both the size and position of a layer in a single call.

Changing a Viewpoint

The **ScrollLayer()** function changes the portion of a super bitmap that is shown by a layer:

```
void ScrollLayer( long dummy, struct Layer *layer, long dx, long dy );
```

This function is most useful with super bitmap layers but can also simulate the effect on other layer types by adding the scroll offset to all future rendering.

Reordering Layers

The layers library provides three function calls for reordering layers:

```
LONG BehindLayer ( long dummy, struct Layer *layer );
LONG UpfrontLayer( long dummy, struct Layer *layer );
LONG MoveLayerInFrontOf( struct Layer *layer_to_move, struct Layer *other_layer );
```

BehindLayer() moves a layer behind all other layers. This function considers any backdrop layers, moving a current layer behind all others except backdrop layers. **UpfrontLayer()** moves a layer in front of all other layers. **MoveLayerInFrontOf()** is used to place a layer at a specific depth, just in front of a given layer.

As areas of simple refresh layers become exposed, due to layer movement or sizing for example, the newly exposed areas have not been drawn into, and need refreshing. The system keeps track of these areas by using a **DamageList**. To update only those areas that need it, the **BeginUpdate() EndUpdate()** functions are called.

```
LONG BeginUpdate( struct Layer *l );
void EndUpdate  ( struct Layer *layer, unsigned long flag );
```

BeginUpdate() saves the pointer to the current clipping rectangles and installs a pointer to a set of **ClipRects** generated from the **DamageList** in the layer structure. To repair the layer, use the graphics rendering routines as if to redraw the entire layer, and the routines will automatically use the new clipping rectangle list. So, only the damaged areas are actually rendered into, saving time.

> *Never Modify the DamageList.* The system generates and maintains the **DamageList** region. All application clipping should be done through the **InstallClipRegion()** function.

To complete the update process call **EndUpdate()** which will restore the original **ClipRect** list.

Sub-Layer Rectangle Operations

The **SwapBitsRastPortClipRect()** routine is for applications that do not want to worry about clipping rectangles.

```
void SwapBitsRastPortClipRect( struct RastPort *rp, struct ClipRect *cr );
```

For instance, you may use The **SwapBitsRastPortClipRect()** to produce a menu without using Intuition. There are two ways to produce such a menu:

1. Create an up-front layer with **CreateUpfrontLayer()**, then render the menu in it. This could use lots of memory and require a lot of (very temporary) ''slice-and-dice'' operations to create all of the clipping rectangles for the existing windows and so on.

2. Use **SwapBitsRastPortClipRect()**, directly on the display drawing area:

   - Render the menu in a back-up area off the display, then lock all of the on-display layers so that no task may use graphics routines to draw over the menu area on the display.

   - Next, swap the on-display bits with the off-display bits, making the menu appear.

   - When finished with the menu, swap again and unlock the layers.

The second method is faster and leaves the clipping rectangles and most of the rest of the window data structures untouched.

> *Warning*: All of the layers must be locked while the menu is visible if you use the second method above. Any task that is using any of the layers for graphics output will be halted while the menu operations are taking place. If, on the other hand, the menu is rendered as a layer, no task need be halted while the menu is up because the lower layers need not be locked.

LAYERS EXAMPLE

For the sake of brevity, the example is a single task. No Layer locking is done. Also note that the routine **myLabelLayer()** is used to redraw a given layer. It is called only when a layer needs refreshing.

```
/* Layers.c
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y layers
** blink FROM LIB:c.o layers.o TO layers LIB LIB:lc.lib LIB:amiga.lib
*/

/* Force use of new variable names to help prevent errors */
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <graphics/gfxbase.h>
#include <graphics/layers.h>
#include <graphics/displayinfo.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/graphics_protos.h>
#include <clib/layers_protos.h>

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

#define L_DELAY  (100)
#define S_DELAY   (50)

#define DUMMY      (0L)

#define RED_PEN    (1)
#define GREEN_PEN  (2)
#define BLUE_PEN   (3)

#define SCREEN_D   (2)
#define SCREEN_W (320)
#define SCREEN_H (200)

/* the starting size of example layers, offsets are used for placement */
#define W_H (50)
#define W_T (5)
#define W_B ((W_T+W_H)-1)
#define W_W (80)
#define W_L ((SCREEN_W/2) - (W_W/2))
#define W_R ((W_L+W_W)-1)

/* size of the superbitmap */
#define SUPER_H SCREEN_H
#define SUPER_W SCREEN_W

/* starting size of the message layer */
#define M_H (10)
#define M_T (SCREEN_H-M_H)
#define M_B ((M_T+M_H)-1)
#define M_W (SCREEN_W)
#define M_L (0)
#define M_R ((M_L+M_W)-1)

struct GfxBase *GfxBase;
struct Library *LayersBase;

/* global constant data for initializing the layers */
LONG  theLayerFlags[3] = { LAYERSUPER, LAYERSMART, LAYERSIMPLE };
UWORD colortable[]     = { 0x000, 0xf44, 0x4f4, 0x44f };


/*
** Clear the layer then draw in a text string.
*/
VOID myLabelLayer(struct Layer *layer, LONG color, UBYTE *string)
{
/* fill layer with color */
SetRast(layer->rp, color);

/* set up for writing text into layer */
SetDrMd(layer->rp,JAM1);
SetAPen(layer->rp,0);
Move(layer->rp, 5, 7);

/* write into layer */
Text(layer->rp, string, strlen(string));
}



/*
** write a message into a layer with a delay.
*/
VOID pMessage(struct Layer *layer, UBYTE *string)
{
Delay(S_DELAY);
myLabelLayer(layer, GREEN_PEN, string);
}
```

```
/*
** write an error message into a layer with a delay.
*/
VOID error(struct Layer *layer, UBYTE *string)
{
myLabelLayer(layer, RED_PEN, string);
Delay(L_DELAY);
}


/*
** do some layers manipulations to demonstrate their abilities.
*/
VOID doLayers(struct Layer *msgLayer, struct Layer *layer_array[])
{
WORD ktr;
WORD ktr_2;

pMessage(msgLayer, "Label all Layers");
myLabelLayer(layer_array[0], RED_PEN,   "Super");
myLabelLayer(layer_array[1], GREEN_PEN, "Smart");
myLabelLayer(layer_array[2], BLUE_PEN,  "Simple");

pMessage(msgLayer, "MoveLayer 1 InFrontOf 0");
if (!MoveLayerInFrontOf(layer_array[1], layer_array[0]))
    error(msgLayer, "MoveLayerInFrontOf() failed.");

pMessage(msgLayer, "MoveLayer 2 InFrontOf 1");
if (!MoveLayerInFrontOf(layer_array[2], layer_array[1]))
    error(msgLayer, "MoveLayerInFrontOf() failed.");

pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN, "Simple");

pMessage(msgLayer, "Incrementally MoveLayers...");
for(ktr = 0; ktr < 30; ktr++)
    {
    if (!MoveLayer(DUMMY, layer_array[1], -1, 0))
        error(msgLayer, "MoveLayer() failed.");
    if (!MoveLayer(DUMMY, layer_array[2], -2, 0))
        error(msgLayer, "MoveLayer() failed.");
    }

pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN, "Simple");

pMessage(msgLayer, "make Layer 0 the UpfrontLayer");
if (!UpfrontLayer(DUMMY, layer_array[0]))
    error(msgLayer, "UpfrontLayer() failed.");

pMessage(msgLayer, "make Layer 2 the BehindLayer");
if (!BehindLayer(DUMMY, layer_array[2]))
    error(msgLayer, "BehindLayer() failed.");

pMessage(msgLayer, "Incrementally MoveLayers again...");
for(ktr = 0; ktr < 30; ktr++)
    {
    if (!MoveLayer(DUMMY, layer_array[1], 0, 1))
        error(msgLayer, "MoveLayer() failed.");
    if (!MoveLayer(DUMMY, layer_array[2], 0, 2))
        error(msgLayer, "MoveLayer() failed.");
    }

pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN, "Simple");

pMessage(msgLayer, "Big MoveLayer");
for(ktr = 0; ktr < 3; ktr++)
    {
    if (!MoveLayer(DUMMY, layer_array[ktr], -layer_array[ktr]->bounds.MinX, 0))
        error(msgLayer, "MoveLayer() failed.");
    }


pMessage(msgLayer, "Incrementally increase size");
```

```
for(ktr = 0; ktr < 5; ktr++)
    {
    for(ktr_2 = 0; ktr_2 < 3; ktr_2++)
        {
        if (!SizeLayer(DUMMY, layer_array[ktr_2], 1, 1))
            error(msgLayer, "SizeLayer() failed.");
        }
    }

pMessage(msgLayer, "Refresh Smart Refresh Layer");
myLabelLayer(layer_array[1], GREEN_PEN, "Smart");
pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN,  "Simple");

pMessage(msgLayer, "Big SizeLayer");
for(ktr = 0; ktr < 3; ktr++)
    {
    if (!SizeLayer(DUMMY,layer_array[ktr],
                SCREEN_W-(layer_array[ktr]->bounds.MaxX)-1,0))
        error(msgLayer, "SizeLayer() failed.");
    }

pMessage(msgLayer, "Refresh Smart Refresh Layer");
myLabelLayer(layer_array[1], GREEN_PEN, "Smart");
pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN,  "Simple");

pMessage(msgLayer, "ScrollLayer down");
for(ktr = 0; ktr < 30; ktr++)
    {
    for(ktr_2 = 0; ktr_2 < 3; ktr_2++)
        {
        ScrollLayer(DUMMY, layer_array[ktr_2], 0, -1);
        }
    }

pMessage(msgLayer, "Refresh Smart Refresh Layer");
myLabelLayer(layer_array[1], GREEN_PEN, "Smart");
pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN,  "Simple");

pMessage(msgLayer, "ScrollLayer up");
for(ktr = 0; ktr < 30; ktr++)
    {
    for(ktr_2 = 0; ktr_2 < 3; ktr_2++)
        {
        ScrollLayer(DUMMY, layer_array[ktr_2], 0, 1);
        }
    }

pMessage(msgLayer, "Refresh Smart Refresh Layer");
myLabelLayer(layer_array[1], GREEN_PEN, "Smart");
pMessage(msgLayer, "Refresh Simple Refresh Layer");
myLabelLayer(layer_array[2], BLUE_PEN,  "Simple");

Delay(L_DELAY);
}


/*
** delete the layer array created by allocLayers().
*/
VOID disposeLayers(struct Layer *msgLayer, struct Layer *layer_array[])
{
WORD ktr;

for (ktr = 0; ktr < 3; ktr++)
    {
    if (layer_array[ktr] != NULL)
        {
        if (!DeleteLayer(DUMMY, layer_array[ktr]))
            error(msgLayer, "Error deleting layer");
        }
    }
}
```

```
/*
** Create some hard-coded layers.  The first must be super-bitmap, with
** the bitmap passed as an argument.  The others must not be super-bitmap.
** The pointers to the created layers are returned in layer_array.
**
** Return FALSE on failure.  On a FALSE return, the layers are
** properly cleaned up.
*/
BOOL allocLayers(struct Layer *msgLayer, struct Layer *layer_array[],
    struct BitMap *super_bitmap, struct Layer_Info *theLayerInfo,
    struct BitMap *theBitMap)
{
WORD ktr;
BOOL create_layer_ok = TRUE;

for (ktr = 0;
     (ktr < 3) && (create_layer_ok);
     ktr++)
    {
    pMessage(msgLayer, "Create BehindLayer");
    if (ktr == 0)
        {
        if ((layer_array[ktr] = CreateBehindLayer(theLayerInfo, theBitMap,
                  W_L+(ktr*30), W_T+(ktr*30), W_R+(ktr*30), W_B+(ktr*30),
                  theLayerFlags[ktr], super_bitmap)) == NULL)
            create_layer_ok = FALSE;
        }
     else
        {
        if ((layer_array[ktr] = CreateBehindLayer(theLayerInfo, theBitMap,
                  W_L+(ktr*30), W_T+(ktr*30), W_R+(ktr*30), W_B+(ktr*30),
                  theLayerFlags[ktr], NULL)) == NULL)
            create_layer_ok = FALSE;
        }

    if (create_layer_ok)
        {
        pMessage(msgLayer, "Fill the RastPort");
        SetRast(layer_array[ktr]->rp, ktr + 1);
        }
    }

if (!create_layer_ok)
    disposeLayers(msgLayer, layer_array);

return(create_layer_ok);
}



/*
** Free the bitmap and all bitplanes created by allocBitMap().
*/
VOID disposeBitMap(struct BitMap *bitmap, LONG depth, LONG width, LONG height)
{
WORD ktr;

if (NULL != bitmap)
    {
    for (ktr = 0; ktr < depth; ktr++)
        {
        if (NULL != bitmap->Planes[ktr])
            FreeRaster(bitmap->Planes[ktr], width, height);
        }

    FreeMem(bitmap, sizeof(*bitmap));
    }
}
```

```
/*
** Allocate and initialize a bitmap structure.
*/
struct BitMap *allocBitMap(LONG depth, LONG width, LONG height)
{
WORD ktr;
BOOL bit_map_failed = FALSE;
struct BitMap *bitmap = NULL;

if (NULL != (bitmap = AllocMem(sizeof(*bitmap),NULL)))
    {
    InitBitMap(bitmap,depth,width,height);

    for (ktr = 0; ktr < depth; ktr++)
        {
        if (NULL == (bitmap->Planes[ktr] = (PLANEPTR)AllocRaster(width,height)))
            bit_map_failed = TRUE;
        else
            BltClear(bitmap->Planes[ktr], RASSIZE(width,height), 1);
        }
    if (bit_map_failed)
        {
        disposeBitMap(bitmap,depth,width,height);
        bitmap = NULL;
        }
    }
return(bitmap);
}

/*
** Set up to run the layers example, doLayers(). Clean up when done.
*/
VOID startLayers(struct Layer_Info *theLayerInfo, struct BitMap *theBitMap)
{
struct Layer  *msgLayer;
struct BitMap *theSuperBitMap;
struct Layer  *theLayers[3] = { NULL, NULL, NULL, };

if (NULL != (msgLayer = CreateUpfrontLayer(theLayerInfo, theBitMap,
                     M_L, M_T, M_R, M_B, LAYERSMART, NULL)))
    {
    pMessage(msgLayer, "Setting up Layers");

    if (NULL != (theSuperBitMap = allocBitMap(SCREEN_D, SUPER_W, SUPER_H)))
        {
        if (allocLayers(msgLayer, theLayers, theSuperBitMap, theLayerInfo, theBitMap))
            {
            doLayers(msgLayer, theLayers);

            disposeLayers(msgLayer, theLayers);
            }
        disposeBitMap(theSuperBitMap, SCREEN_D, SUPER_W, SUPER_H);
        }
    if (!DeleteLayer(DUMMY, msgLayer))
        error(msgLayer, "Error deleting layer");
    }
}

/*
** Set up a low-level graphics display for layers to work on.  Layers
** should not be built directly on Intuition screens, use a low-level
** graphics view.  If you need mouse or other events for the layers
** display, you have to get them directly from the input device.  The
** only supported method of using layers library calls with Intuition
** (other than the InstallClipRegion() call) is through Intuition windows.
**
** See graphics primitives chapter for details on creating and using the
** low-level graphics calls.
*/
VOID runNewView(VOID)
{
struct View        theView;
struct View       *oldview;
struct ViewPort    theViewPort;
struct RasInfo     theRasInfo;
struct ColorMap   *theColorMap;
```

```
struct Layer_Info *theLayerInfo;
struct BitMap     *theBitMap;
UWORD             *colorpalette;
WORD               ktr;

/* save current view, to be restored when done */
if (NULL != (oldview = GfxBase->ActiView))
    {
    /* get a LayerInfo structure */
    if (NULL != (theLayerInfo = NewLayerInfo()))
        {
        if (NULL != (theColorMap = GetColorMap(4)))
            {
            colorpalette = (UWORD *)theColorMap->ColorTable;
            for(ktr = 0; ktr < 4; ktr++)
                *colorpalette++ = colortable[ktr];

            if (NULL != (theBitMap = allocBitMap(SCREEN_D, SCREEN_W, SCREEN_H)))
                {
                InitView(&theView);
                InitVPort(&theViewPort);

                theView.ViewPort = &theViewPort;

                theViewPort.DWidth   = SCREEN_W;
                theViewPort.DHeight  = SCREEN_H;
                theViewPort.RasInfo  = &theRasInfo;
                theViewPort.ColorMap = theColorMap;

                theRasInfo.BitMap   = theBitMap;
                theRasInfo.RxOffset = 0;
                theRasInfo.RyOffset = 0;
                theRasInfo.Next     = NULL;

                MakeVPort(&theView, &theViewPort);
                MrgCop(&theView);
                LoadView(&theView);
                WaitTOF();

                startLayers(theLayerInfo, theBitMap);

                /* put back the old view, wait for it to become
                ** active before freeing any of our display
                */
                LoadView(oldview);
                WaitTOF();

                /* free dynamically created structures */
                FreeVPortCopLists(&theViewPort);
                FreeCprList(theView.LOFCprList);

                disposeBitMap(theBitMap, SCREEN_D, SCREEN_W, SCREEN_H);
                }
            FreeColorMap(theColorMap);       /* free the color map */
            }
        DisposeLayerInfo(theLayerInfo);
        }
    }
}

/*
** Open the libraries used by the example.  Clean up when done.
*/
VOID main(int argc, char **argv)
{
if (NULL != (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",33L)))
    {
    if (NULL != (LayersBase = OpenLibrary("layers.library",33L)))
        {
        runNewView();

        CloseLibrary((struct Library *)LayersBase);
        }
    CloseLibrary((struct Library *)GfxBase);
    }
}
```

Regions

Regions allow the application to install clipping rectangles into layers. A clipping rectangle is a rectangular area into which the graphics routines will draw. All drawing that would fall outside of that rectangular area is clipped (not rendered).

User clipping regions are linked lists of clipping rectangles created by an application program through the graphics library routines described below. By combining together various clipping rectangles, any arbitrary clipping shape can be created. Once the region is set up, you use the layers library call **InstallClipRegion()** to make the clipping region active in a layer.

Regions are safe to use with layers created by Intuition (i.e., windows). The following table describes the routines available for the creation, manipulation and use of regions.

Table 30-4: Functions Used with Regions

Routine	Library	Description
InstallClipRegion()	Layers	Add a clipping region to a layer.
NewRegion()	Graphics	Create a new, empty region.
DisposeRegion()	Graphics	Dispose of an existing region and its rectangles.
AndRectRegion()	Graphics	*And* a rectangle into a region.
OrRectRegion()	Graphics	*Or* a rectangle into a region.
XorRectRegion()	Graphics	*Exclusive-or* a rectangle into a region.
ClearRectRegion()	Graphics	Clear a rectangular portion of a region.
AndRegionRegion()	Graphics	*And* two regions together.
OrRegionRegion()	Graphics	*Or* two regions together.
XorRegionRegion()	Graphics	*Exclusive-or* two regions together.
ClearRegion()	Graphics	Clear a region.

With these functions, the application can selectively update a custom-shaped part of a layer without disturbing any of the other layers that might be present.

> *Never Modify the DamageList of a Layer Directly.* Use the routine **InstallClipRegion()** to add clipping to the layer. The regions installed by **InstallClipRegion()** are independent of the layer's **DamageList** and use of user clipping regions will not interfere with optimized window refreshing.

> *Do Not Modify A Region After It Has Been Added.* After a region has been added with **InstallClipRegion()**, the program may not modify it until the region has been removed with another call to **InstallClipRegion()**.

CREATING AND DELETING REGIONS

You allocate a **Region** data structure with the **NewRegion()** call.

```
struct Region *NewRegion( void );
```

The **NewRegion()** function allocates and initializes a **Region** structure that has no drawable areas defined in it. If the application draws through a new region, nothing will be drawn as the region is empty. The application must add rectangles to the region before any graphics will appear.

Use **DisposeRegion()** to free the **Region** structure when you are done with it.

```
void DisposeRegion( struct Region *region );
```

DisposeRegion() returns all memory associated with a region to the system and deallocates all rectangles that have been linked to it.

> *Don't Forget to Free Your Rectangles.* All of the functions that add rectangles to the region make copies of the rectangles. If the program allocates a rectangle, then adds it to a region, it still must deallocate the rectangle. The call to **DisposeRegion()** will not deallocate rectangles *explicitly allocated* by the application.

INSTALLING REGIONS

Use the function **InstallClipRegion()** to install the region.

```
struct Region *InstallClipRegion( struct Layer *layer, struct Region *region );
```

This installs a transparent clipping region to a layer. All subsequent graphics calls will be clipped to this region. The region must be removed with a second call to **InstallClipRegion()** before removing the layer.

```
/*
** Sample installation and removal of a clipping region
*/
register struct Region     *new_region ;
register struct Region     *old_region ;

/* If the application owns the layer and has not installed a region,
** old_region will return NULL here.
*/
old_region = InstallClipRegion(win->WLayer, new_region);

/* draw into the layer or window */

if (NULL != (old_region = InstallClipRegion(win->WLayer, old_region)))
    {
    /* throw the used region away.  This region could be saved and
    ** used again later, if desired by the application.
    */
    DisposeRegion(new_region) ;
    }
```

A Warning About InstallClipRegion(). The program must not call **InstallClipRegion()** inside of a **Begin/EndRefresh()** or **Begin/EndUpdate()** pair. The following code segment shows how to modify the user clipping region when using these calls. See the Autodoc for **BeginRefresh()** for more details.

```
register struct Region     *new_region ;
register struct Region     *old_region ;

/* you have to have already setup the new_region and old_region */

BeginRefresh(window);
/* draw through the damage list */
/* into the layer or window */
EndRefresh(window, FALSE);             /* keep the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the damage list and the new_region */
/* into the layer or window */
EndRefresh(window, FALSE);             /* keep the damage list */

/* put back the old region */
new_region = InstallClipRegion(win->WLayer, old_region);

BeginRefresh(window);
EndRefresh(window, TRUE);              /* remove the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the new_region only into the layer or window */
EndRefresh(window, FALSE);

/* finally get rid of the new region, old_region still installed */
if (NULL != (new_region = InstallClipRegion(win->WLayer, old_region)))
    DisposeRegion(new_region) ;
```

CHANGING A REGION

Regions may be modified by performing logical operations with rectangles, or with other regions.

Reuse Your Rectangles. In all of the rectangle and region routines the clipping rectangle is copied into the region. This means that a single clipping rectangle (**Rectangle** structure) may be used many times by simply changing the x and y values. The application need *not* create a new instance of the **Rectangle** structure for each rectangle added to a region.

For instance:

```
extern struct Region *RowRegion;  /* created elsewhere */

WORD ktr;
struct Rectangle rect;

for (ktr = 1; ktr < 6; ktr++)
    {
    rect.MinX = 50;
    rect.MaxX = 315;
    rect.MinY = (ktr * 10) - 5;
    rect.MaxY = (ktr * 10);

    if (!OrRectRegion(RowRegion, &rect))
        clean_exit(RETURN_WARN);
    }
```

Rectangles and Regions

There are four functions for changing a region through logical operations with a rectangle.

```
BOOL OrRectRegion   ( struct Region *region, struct Rectangle *rectangle );
void AndRectRegion  ( struct Region *region, struct Rectangle *rectangle );
BOOL XorRectRegion  ( struct Region *region, struct Rectangle *rectangle );
BOOL ClearRectRegion( struct Region *region, struct Rectangle *rectangle );
```

OrRectRegion() modifies a region structure by *or*'ing a clipping rectangle into the region. When the application draws through this region (assuming that the region was originally empty), only the pixels within the clipping rectangle will be affected. If the region already has drawable areas, they will still exist, this rectangle is added to the drawable area.

AndRectRegion() modifies the region structure by *and*'ing a clipping rectangle into the region. Only those pixels that were already drawable and within the rectangle will remain drawable, any that are outside of it will be clipped in future.

XorRectRegion() applies the rectangle to the region in an *exclusive-or* mode. Within the given rectangle, any areas that were drawable become clipped, any areas that were clipped become drawable. Areas outside of the rectangle are not affected.

ClearRectRegion() clears the rectangle from the region. Within the given rectangle, any areas that were drawable become clipped. Areas outside of the rectangle are not affected.

Regions and Regions

As with rectangles and regions, there are four layers library functions for combining regions with regions:

```
BOOL AndRegionRegion( struct Region *srcRegion, struct Region *destRegion );
BOOL OrRegionRegion ( struct Region *srcRegion, struct Region *destRegion );
BOOL XorRegionRegion( struct Region *srcRegion, struct Region *destRegion );
void ClearRegion    ( struct Region *region );
```

AndRegionRegion() performs a logical *and* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever the regions drawable areas overlap. That is, where there are drawable areas in both region 1 *and* region 2, there will be drawable areas left in the result region.

OrRegionRegion() performs a logical *or* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever there are drawable areas in either region. That is, where there are drawable areas in either region 1 *or* region 2, there will be drawable areas left in the result region.

XorRegionRegion() performs a logical *exclusive-or* operation on the two regions, leaving the result in the second region. The operation leaves drawable areas wherever there are drawable areas in either region but not both. That is, where there are drawable areas in either region 1 *or* region 2, there will be drawable areas left in the result region. But where there are drawable areas in both region 1 *and* region 2, there will **not** be drawable areas left in the result region.

ClearRegion() puts the region back to the same state it was in when the region was created with **NewRegion()**, that is, no areas are drawable.

REGIONS EXAMPLE

The following example shows the use of the layers library call **InstallClipRegion()**, as well as simple use of the graphics library regions functions. Be aware that it uses Release 2 functions for opening and closing Intuition windows.

```
/* clipping.c
**
** SAS/C 5.10a
** lc -b1 -cfist -v -y clipping
** blink FROM LIB:c.o clipping.o TO clipping LIB LIB:lc.lib LIB:amiga.lib
*/

/* Force use of new variable names to help prevent errors */
#define INTUI_V36_NAMES_ONLY

#include <exec/types.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/displayinfo.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/layers_protos.h>

#include <stdio.h>

#ifdef LATTICE
int CXBRK(void)     { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void)  { return(0); }  /* really */
#endif

#define MY_WIN_WIDTH  (300)
#define MY_WIN_HEIGHT (100)

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Library *LayersBase;

/*
** unclipWindow()
**
** Used to remove a clipping region installed by clipWindow() or
** clipWindowToBorders(), disposing of the installed region and
** reinstalling the region removed.
*/
void unclipWindow(struct Window *win)
{
struct Region     *old_region;

/* Remove any old region by installing a NULL region,
** then dispose of the old region if one was installed.
*/
if (NULL != (old_region = InstallClipRegion(win->WLayer, NULL)))
    DisposeRegion(old_region);
}

/*
** clipWindow()
** Clip a window to a specified rectangle (given by upper left and
** lower right corner.)  the removed region is returned so that it
** may be re-installed later.
*/
struct Region *clipWindow(struct Window *win,
    LONG minX, LONG minY, LONG maxX, LONG maxY)
{
struct Region    *new_region;
struct Rectangle  my_rectangle;
```

```
/* set up the limits for the clip */
my_rectangle.MinX = minX;
my_rectangle.MinY = minY;
my_rectangle.MaxX = maxX;
my_rectangle.MaxY = maxY;

/* get a new region and OR in the limits. */
if (NULL != (new_region = NewRegion()))
    {
    if (FALSE == OrRectRegion(new_region, &my_rectangle))
        {
        DisposeRegion(new_region);
        new_region = NULL;
        }
    }

/* Install the new region, and return any existing region.
** If the above allocation and region processing failed, then
** new_region will be NULL and no clip region will be installed.
*/
return(InstallClipRegion(win->WLayer, new_region));
}

/*
** clipWindowToBorders()
** clip a window to its borders.
** The removed region is returned so that it may be re-installed later.
*/
struct Region *clipWindowToBorders(struct Window *win)
{
return(clipWindow(win, win->BorderLeft, win->BorderTop,
    win->Width - win->BorderRight - 1, win->Height - win->BorderBottom - 1));
}

/*
** Wait for the user to select the close gadget.
*/
VOID wait_for_close(struct Window *win)
{
struct IntuiMessage *msg;
SHORT done;

done = FALSE;
while (FALSE == done)
    {
    /* we only have one signal bit, so we do not have to check which
    ** bit broke the Wait().
    */
    Wait(1L << win->UserPort->mp_SigBit);

    while ( (FALSE == done) &&
            (NULL != (msg = (struct IntuiMessage *)GetMsg(win->UserPort))))
        {
        /* use a switch statement if looking for multiple event types */
        if (msg->Class == IDCMP_CLOSEWINDOW)
            done = TRUE;

        ReplyMsg((struct Message *)msg);
        }
    }
}

/*
** Simple routine to blast all bits in a window with color three to show
** where the window is clipped.  After a delay, flush back to color zero
** and refresh the window borders.
*/
VOID draw_in_window(struct Window *win, UBYTE *message)
{
printf("%s...", message); fflush(stdout);
SetRast(win->RPort, 3);
Delay(200);
SetRast(win->RPort, 0);
RefreshWindowFrame(win);
printf("done\n");
}
```

```
/*
** Show drawing into an unclipped window, a window clipped to the
** borders and a window clipped to a random rectangle.  It is possible
** to clip more complex shapes by AND'ing, OR'ing and exclusive-OR'ing
** regions and rectangles to build a user clip region.
**
** This example assumes that old regions are not going to be re-used,
** so it simply throws them away.
*/
VOID clip_test(struct Window *win)
{
struct Region    *old_region;

draw_in_window(win,"Window with no clipping");

/* if the application has never installed a user clip region,
** then old_region will be NULL here.  Otherwise, delete the
** old region (you could save it and re-install it later...)
*/
if (NULL != (old_region = clipWindowToBorders(win)))
    DisposeRegion(old_region);
draw_in_window(win,"Window clipped to window borders");
unclipWindow(win);

/* here we know old_region will be NULL, as that is what we
** installed with unclipWindow()...
*/
if (NULL != (old_region = clipWindow(win,20,20,100,50)))
    DisposeRegion(old_region);
draw_in_window(win,"Window clipped from (20,20) to (100,50)");
unclipWindow(win);

wait_for_close(win);
}



/*
** Open and close resources, call the test routine when ready.
*/
VOID main(int argc, char **argv)
{
struct Window *win;

if (NULL != (IntuitionBase =
        (struct IntuitionBase *)OpenLibrary("intuition.library",37)))
    {
    if (NULL != (GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",37)))
        {
        if (NULL != (LayersBase = OpenLibrary("layers.library",37)))
            {
            if (NULL != (win = OpenWindowTags(NULL,
                                WA_Width,       MY_WIN_WIDTH,
                                WA_Height,      MY_WIN_HEIGHT,
                                WA_IDCMP,       IDCMP_CLOSEWINDOW,
                                WA_CloseGadget, TRUE,
                                WA_DragBar,     TRUE,
                                WA_Activate,    TRUE,
                                TAG_END)))
                {
                clip_test(win);

                CloseWindow(win);
                }
            CloseLibrary(LayersBase);
            }
        CloseLibrary((struct Library *)GfxBase);
        }
    CloseLibrary((struct Library *)IntuitionBase);
    }
}
```

Function Reference

The following are brief descriptions of the layers library functions and related routines from the graphics library. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 30-5: Layers Library Functions

Function	Description
NewLayerInfo()	Allocating a **Layer_Info** structure.
DisposeLayerInfo()	Deallocating a **Layer_Info** structure.
CreateUpfrontLayer()	Make a new layer in front of others.
CreateBehindLayer()	Make a new layer behind others.
DeleteLayer()	Remove and delete an existing layer.
MoveLayer()	Change the position (not depth) of a layer.
SizeLayer()	Change the size of a layer.
ScrollLayer()	Change the internal coordinates of a layer.
BehindLayer()	Depth arrange a layer behind others.
UpfrontLayer()	Depth arrange a layer in front of others.
MoveLayerInFrontOf()	Depth arrange a layer to a specific position.
WhichLayer()	Find the frontmost layer at a position.
SwapBitsRastPortClipRect()	Fast, non-layered and non-damaging display operation.
BeginUpdate()	Synchronize optimized refreshing for layer.
EndUpdate()	End optimized layer refresh.
LockLayer()	Lock out rendering in a single layer.
UnlockLayer()	Release **LockLayer()** lock.
LockLayers()	Lock out rendering in all layers of a display.
UnlockLayers()	Release **LockLayers()** lock.
LockLayerInfo()	Gain exclusive access to the display's layers.
UnlockLayerInfo()	Release **LockLayerInfo()** lock.
InstallClipRegion()	Add a clipping region to a layer.

The following routines from graphics library are also required for certain layers library functions:

Routine	Description
LockLayerRom()	Same as **LockLayer()**, from layers library.
UnlockLayerRom()	Release **LockLayerRom()** lock.
AttemptLockLayerRom()	Lock layer only if it is immediately available.
NewRegion()	Create a new, empty region.
DisposeRegion()	Dispose of an existing region and its rectangles.
AndRectRegion()	AND a rectangle into a region.
OrRectRegion()	OR a rectangle into a region.
XorRectRegion()	Exclusive-OR a rectangle into a region.
ClearRectRegion()	Clear a region.
AndRegionRegion()	AND two regions together.
OrRegionRegion()	OR two regions together.
XorRegionRegion()	Exclusive-OR two regions together.
ClearRegion()	Clear a region.

Chapter 31
COMMODITIES EXCHANGE LIBRARY

This chapter describes Commodities Exchange, the library of routines used to add a custom input handler to the Amiga. With Commodities Exchange, any program function can be associated with key combinations or other input events *globally* allowing the creation utility programs that run in the background for all tasks.

Custom Input Handlers

The input.device has a hand in almost all user input on the Amiga. It gathers input events from the keyboard, the gameport (mouse), and several other sources, into one input "stream". Special programs called input event handlers intercept input events along this stream, examining and sometimes changing the input events. Both Intuition and the console device use input handlers to process user input.

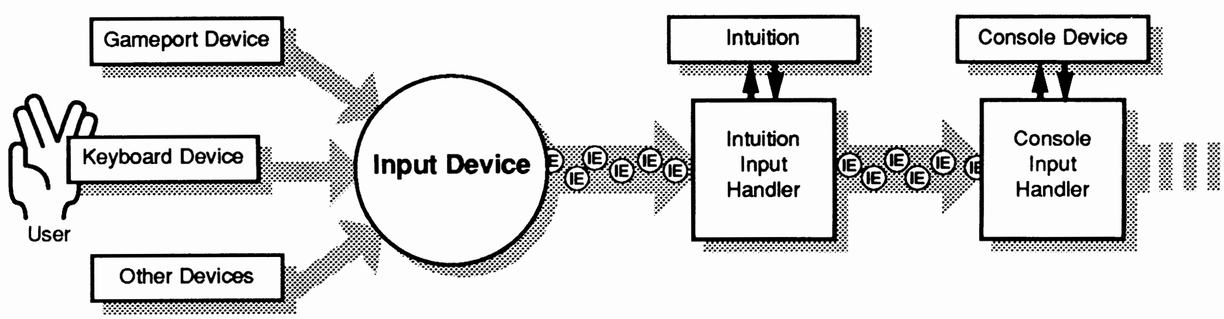


Figure 31-1: The Amiga Input Stream

Using the input.device, a program can introduce its own custom handler into the chain of input handlers at almost any point in the chain. "Hot key" programs, shell pop-up programs, and screen blankers all commonly use custom input handlers to monitor user input before it gets to the Intuition input handler.

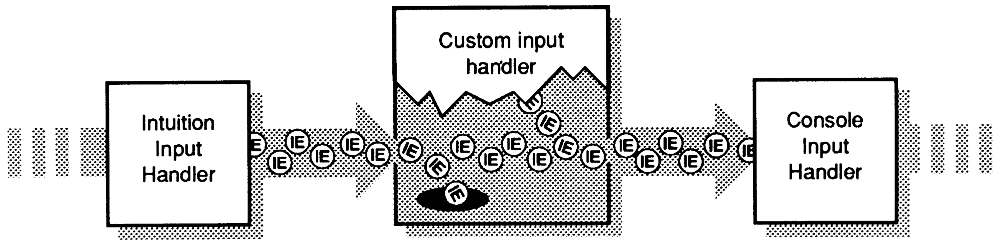


Figure 31-2: A Custom Input Handler

Custom input handlers do have their drawbacks, however. Not only are these handlers hard to program, but because there is no standard way to implement and control them, multiple handlers often do not work well together. Their antisocial behavior can result in load order dependencies and incompatibilities between different custom input handlers. Even for the expert user, having several custom input handlers coexist peacefully can be next to impossible.

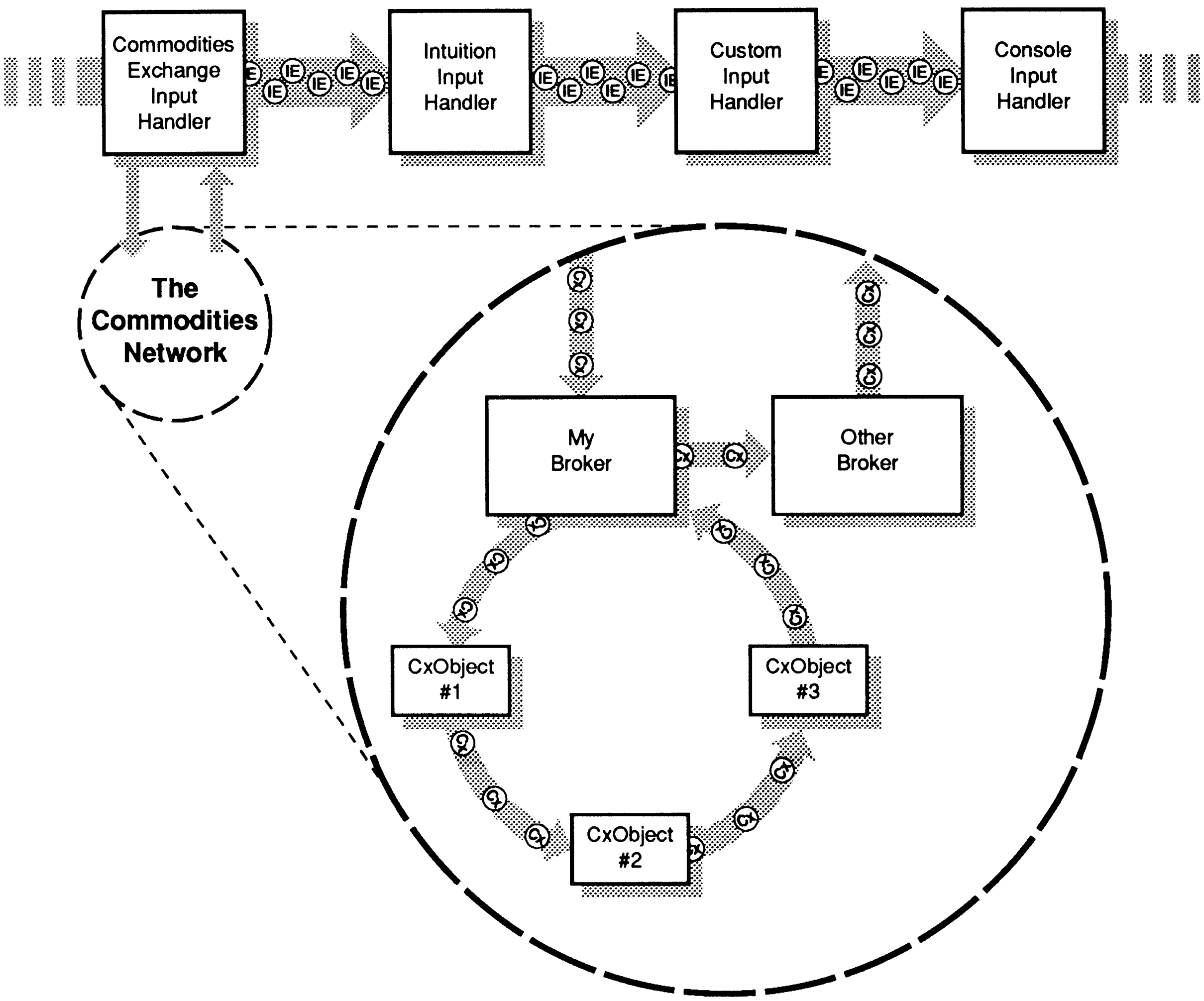


Figure 31-3: The Commodities Network

Commodities Exchange eliminates these problems by providing a simple, standardized way to program and control custom input handlers. It is divided into three parts: an Exec library, a controller program, and some amiga.lib functions.

The Exec library is called *commodities.library*. When it is first opened, commodities.library establishes a single input handler just before Intuition in the input chain. When this input handler receives an input event, it creates a **CxMessage** (Commodities Exchange Message) corresponding to the input event, and diverts the **CxMessage** through the network of Commodities Exchange input handlers (Figure 31-3).

These handlers are made up of trees of different **CxObjects** (Commodities Exchange Objects), each of which performs a simple operation on the **CxMessages**. Any **CxMessages** that exit the network are returned to the input.device's input stream as input events.

Through function calls to the commodities.library, an application can install a custom input handler. A Commodities Exchange application, sometimes simply referred to as a commodity, uses the **CxObject** primitives to do things such as filter certain **CxMessages**, translate **CxMessages**, signal a task when a **CxObject** receives a **CxMessage**, send a message when a **CxObject** receives a **CxMessage**, or if necessary, call a custom function when a **CxObject** receives a **CxMessage**.

The controller program is called Commodities Exchange. The user can monitor and control all the currently running Commodities Exchange applications from this one program. The user can enable and disable a commodity, kill a commodity, or, if the commodity has a window, ask the commodity to show or hide its window. When the user requests any of these actions, the controller program sends the commodity a message, telling it which action to perform.

The third component of Commodities Exchange is its scanned library functions. These functions are part of the amiga.lib scanned library. They do a lot of the work involved with parsing command lines and Tool Types.

Commodities Exchange is ideal for programs like hot keys/pop ups, screen blankers, and mouse blankers that need to monitor all user input. *Commodities Exchange should never be used as an alternate method of receiving user input for an application.* Other applications depend on getting user input in some form or another from the input stream. A greedy program that diverts input to itself rather than letting the input go to where the user expects it can seriously confuse the user, not to mention compromise the advantages of multitasking.

CxObjects

CxObjects are the basic building blocks used to construct a commodity. A commodity uses **CxObjects** to take care of all manipulations of **CxMessages**. When a **CxMessage** "arrives" at a **CxObject**, that **CxObject** carries out its primitive action and then, if it has not deleted the **CxMessage**, it passes the **CxMessage** on to the next **CxObject**. A commodity links together **CxObjects** into a tree, organizing these simple action objects to perform some higher function.

A **CxObject** is in one of two states, active or inactive. An active **CxObject** performs its primitive action every time it receives a **CxMessage**. If a **CxObject** is inactive, **CxMessages** bypass it, continuing to the **CxObject** that follows the inactive one. By default, all **CxObjects** except the type called brokers are created in the active state.

Currently, there are seven types of **CxObjects** (Table 31-1).

Object Type	Purpose
Broker	Registers a new commodity with the commodity network
Filter	Accepts or rejects input events based on criteria set up by the application
Sender	Sends a message to a message port
Translate	Replaces the input event with a different one
Signal	Signals a task
Custom	Calls a custom function provided by the commodity
Debug	Sends debug information out the serial port

Table 31-1: Commodities Exchange Object Types

Installing A Broker Object

The Commodities Exchange input handler maintains a master list of **CxObjects** to which it diverts input events using **CxMessages**. The **CxObjects** in this master list are a special type of **CxObject** called *brokers*. The only thing a broker **CxObject** does is divert **CxMessages** to its own personal list of **CxObjects**. A commodity creates a broker and attaches other **CxObjects** to it. These attached objects take care of the actual input handler related work of the commodity and make up the broker's personal list.

The first program listing, Broker.c, is a very simple example of a working commodity. It serves only to illustrate the basics of a commodity, not to actually perform any useful function. It shows how to set up a broker and process commands from the controller program.

Besides opening commodities.library and creating an Exec message port, setting up a commodity requires creating a broker. The function **CxBroker()** creates a broker and adds it to the master list.

```
CxObj *CxBroker(struct NewBroker *nb, long *error);
```

CxBroker()'s first argument is a pointer to a **NewBroker** structure:

```
struct NewBroker {
   BYTE     nb_Version;   /* There is an implicit pad byte after this BYTE */
   BYTE     *nb_Name;
   BYTE     *nb_Title;
   BYTE     *nb_Descr;
   SHORT    nb_Unique;
   SHORT    nb_Flags;
   BYTE     nb_Pri;       /* There is an implicit pad byte after this BYTE */
   struct   MsgPort   *nb_Port;
   WORD     nb_ReservedChannel;  /* Unused, make zero for future compatibility */
};
```

Commodities Exchange gets all the information it needs about the broker from this structure. **NewBroker**'s **nb_Version** field contains the version number of the **NewBroker** structure. This should be set to NB_VERSION which is defined in *<libraries/commodities.h>*. The **nb_Name**, **nb_Title**, and **nb_Descr** point to strings which hold the name, title, and description of the broker. The two bit fields, **nb_Unique** and **nb_Flags**, toggle certain features of Commodities Exchange based on their values. They are discussed in detail later in this chapter.

The **nb_Pri** field contains the broker's priority. Commodities Exchange inserts the broker into the master list based on this number. Higher priority brokers get **CxMessages** before lower priority brokers.

CxBroker()'s second argument is a pointer to a LONG. If this pointer is not NULL, **CxBroker()** fills in this field with one of the following error return codes from *<libraries/commodities.h>*:

```
CBERR_OK        0       /* No error                          */
CBERR_SYSERR    1       /* System error , no memory, etc     */
CBERR_DUP       2       /* uniqueness violation              */
CBERR_VERSION   3       /* didn't understand nb_VERSION      */
```

Once the broker object is created with **CxBroker()**, it must be activated with **ActivateCxObject()**.

```
oldactivationvalue = LONG ActivateCxObj(CxObj *co, long newactivationvalue);
```

After successfully completing the initial set up and activating the broker, a commodity can begin its input processing loop waiting for **CxMessages** to arrive.

CxMessages

There are actually two types of **CxMessages**. The first, CXM_IEVENT, corresponds to an input event and travels through the Commodities Exchange network. The other type, CXM_COMMAND, carries a command to a commodity. A CXM_COMMAND normally comes from the controller program and is used to pass user commands on to a commodity. A commodity receives these commands through an Exec message port that the commodity sets up before it calls **CxBroker()**. The **NewBroker**'s **nb_Port** field points to this message port. A commodity can tell the difference between the two types of **CxMessages** by calling the **CxMsgType()** function.

```
ULONG  CxMsgType( CxMsg *cxm );
UBYTE *CxMsgData( CxMsg *cxm );
LONG   CxMsgID  ( CxMsg *cxm );
```

A **CxMessage** not only has a type, it can also have a data pointer as well as an ID associated with it. The data associated with a CXM_IEVENT **CxMessage** is an **InputEvent** structure. By using the **CxMsgData()** function, a commodity can obtain a pointer to the corresponding **InputEvent** of a CXM_IEVENT message. Commodities Exchange gives an ID of zero to any CXM_IEVENT **CxMessage** that it introduces to the Commodities network but certain **CxObjects** can assign an ID to them.

For a CXM_COMMAND **CxMessages**, the data pointer is generally not used but the ID specifies a command passed to the commodity from the user operating the controller program. The **CxMsgID()** macro extracts the ID from a **CxMessage**.

A SIMPLE COMMODITY EXAMPLE

The example below, Broker.c, receives input from one source, the controller program. The controller program sends a **CxMessage** each time the user clicks its Enable, Disable, or Kill gadgets. Using the **CxMsgID()** function, the commodity finds out what the command is and executes it.

```
/* broker.c - Simple skeletal example of opening a broker
 * compiled with SASC 5.10
 * LC -b0 -cfist -v broker.c
 * Blink FROM LIB:c.o,broker.o TO broker LIBRARY LIB:LC.lib,LIB:Amiga.lib
 */
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
```

```
#include <clib/alib_stdio_protos.h>
#include <clib/commodities_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

void main(void);
void ProcessMsg(void);

struct Library *CxBase;

CxObj *broker;
struct MsgPort *broker_mp;
ULONG cxsigflag;

struct NewBroker newbroker = {
    NB_VERSION,   /* nb_Version - Version of the NewBroker structure */
    "RKM broker", /* nb_Name - Name Commodities uses to identify this commodity */
    "Broker",     /* nb_Title - Title of commodity that appears in CXExchange */
    "A simple example of a broker",  /* nb_Descr - Description of the commodity */
    0,            /* nb_Unique - Tells CX not to launch another commodity with same name */
    0,            /* nb_Flags - Tells CX if this commodity has a window */
    0,            /* nb_Pri - This commodity's priority */
    0,            /* nb_Port - MsgPort CX talks to */
    0             /* nb_ReservedChannel - reserved for later use */
};


void main(void)
{
    CxMsg *msg;

    /* Before bothering with anything else, open the library */
    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        /* Commodities talks to a Commodities application through */
        /* an Exec Message port, which the application provides   */
        if (broker_mp = CreateMsgPort())
        {
            newbroker.nb_Port = broker_mp;

            /* The commodities.library function CxBroker() adds a borker to the
             * master list.  It takes two arguments, a pointer to a NewBroker
             * structure and a pointer to a LONG.  The NewBroker structure contains
             * information to set up the broker.  If the second argument is not
             * NULL, CxBroker will fill it in with an error code.
             */
            if (broker = CxBroker(&newbroker, NULL))
            {
                cxsigflag = 1L << broker_mp->mp_SigBit;

                /* After it's set up correctly, the broker has to be activated */
                ActivateCxObj(broker, 1L);

                /* the main processing loop */
                ProcessMsg();

                /* It's time to clean up.  Start by removing the broker from the
                 * Commodities master list.  The DeleteCxObjAll() function will
                 * take care of removing a CxObject and all those connected
                 * to it from the Commodities network
                 */
                DeleteCxObj(broker);

                /* Empty the port of CxMsgs */
                while(msg = (CxMsg *)GetMsg(broker_mp))
                  ReplyMsg((struct Message *)msg);
            }
            DeletePort(broker_mp);
        }
        CloseLibrary(CxBase);
    }
}
```

```
void ProcessMsg(void)
{
    CxMsg *msg;

    ULONG sigrcvd, msgid, msgtype;
    LONG returnvalue = 1L;

    while (returnvalue)
    {
        /* wait for something to happen */
        sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

        /* process any messages */
        while(msg = (CxMsg *)GetMsg(broker_mp))
        {
            /* Extract necessary information from the CxMessage and return it */
            msgid = CxMsgID(msg);
            msgtype = CxMsgType(msg);
            ReplyMsg((struct Message *)msg);

            switch(msgtype)
            {
                case CXM_IEVENT:
                    /* Shouldn't get any of these in this example */
                    break;
                case CXM_COMMAND:
                    /* Commodities has sent a command */
                    printf("A command: ");
                    switch(msgid)
                    {
                        case CXCMD_DISABLE:
                            printf("CXCMD_DISABLE\n");
                            /* The user clicked Commodities Exchange disable
                             * gadget better disable
                             */
                            ActivateCxObj(broker, 0L);
                            break;
                        case CXCMD_ENABLE:
                            /* user clicked enable gadget */
                            printf("CXCMD_ENABLE\n");
                            ActivateCxObj(broker, 1L);
                            break;
                        case CXCMD_KILL:
                            /* user clicked kill gadget, better quit */
                            printf("CXCMD_KILL\n");
                            returnvalue = 0L;
                            break;
                    }
                    break;
                default:
                    printf("Unknown msgtype\n");
                    break;
            }
        }
        /* Test to see if user tried to break */
        if (sigrcvd & SIGBREAKF_CTRL_C)
        {
            returnvalue = 0L;
            printf("CTRL C signal break\n");
        }
    }
}
```

Notice that Broker.c uses Ctrl-C as a break key. The break key for any commodity should be Ctrl-C.

CONTROLLER COMMANDS

The commands that a commodity can receive from the controller program (as defined in <*libraries/commodities.h*>) are:

```
CXCMD_DISABLE     /* please disable yourself       */
CXCMD_ENABLE      /* please enable yourself        */
CXCMD_KILL        /* go away for good              */
CXCMD_APPEAR      /* open your window, if you can  */
CXCMD_DISAPPEAR   /* hide your window              */
```

The CXCMD_DISABLE, CXCMD_ENABLE, and CXCMD_KILL commands correspond to the similarly named controller program gadgets, Disable, Enable, and Kill; CXCMD_APPEAR and CXCMD_DISAPPEAR correspond to the controller program gadgets, Show and Hide. These gadgets are ghosted in Broker.c because it has no window (It doesn't make much sense to give the user a chance to click the Show and Hide gadgets). In order to do this, Broker.c has to tell Commodities Exchange to ghost these gadgets. When **CxBroker()** sets up a broker, it looks at the **NewBroker.nb_Flags** field to see if the COF_SHOW_HIDE bit (from <*libraries/commodities.h*>) is set. If it is, the "Show" and "Hide" gadgets for this broker will be selectable. Otherwise they are ghosted and disabled.

SHUTTING DOWN THE COMMODITY

Shutting down a commodity is easy. After replying to all **CxMessages** waiting at the broker's message port, a commodity can delete its **CxObjects**. The **DeleteCxObj()** function removes a single **CxObject** from the Commodities network. **DeleteCxObjectAll()** removes multiple objects.

```
void DeleteCxObj( CxObj *co );
void DeleteCxObjAll( CxObj *delete_co );
```

If a commodity has a lot of **CxObjects**, deleting each individually can be a bit tedious. **DeleteCxObjAll()** will delete a **CxObject** and any other **CxObjects** that are attached to it. The HotKey.c example given later in this chapter uses this function to delete all its **CxObjects**. A commodity that uses **DeleteCxObjAll()** to delete all its **CxObjects** should make sure that they are all connected to the main one. (See the "Connecting CxObjects" section below.)

After deleting its **CxObjects**, a commodity must take care of any **CxMessages** that might have arrived at the message port just before the commodity deleted its objects.

```
while(msg = (CxMsg *)GetMsg(broker_mp))
    ReplyMsg((struct Message *)msg);
```

Commodity Tool Types

A goal of Commodities Exchange is to improve user control over input handlers. One way in which it accomplishes this goal is through the use of standard icon Tool Types. The user will expect commodities to recognize the set of standard Tool Types:

CX_PRIORITY
CX_POPUP
CX_POPKEY

CX_PRIORITY lets the user set the priority of a commodity. The string "CX_PRIORITY=" is a number from -128 to 127. The higher the number, the higher the priority of the commodity, giving it access to input events before lower priority commodities. All commodities should recognize CX_PRIORITY.

CX_POPUP and CX_POPKEY are only relevant to commodities with a window. The string "CX_POPUP=" should be followed by a "yes" or "no", telling the commodity if it should or shouldn't show its window when it is first launched. CX_POPKEY is followed by a string describing the key to use as a hot key for making the commodity's window appear (pop up). The description string for CX_POPKEY describes an input event. The specific format of the string is discussed in the next section ("Filter Objects and the Input Description String").

Commodities Exchange's support library functions simplify parsing arguments from either the Workbench or the Shell (CLI). A Workbench launched commodity gets its arguments directly from the Tool Types in the commodity's icon. Shell launched commodities get their arguments from the command line, but these arguments look exactly like the Tool Types from the commodity's icon. For example, the following command line sets the priority of a commodity called HotKey to 5:

```
HotKey "CX_PRIORITY=5"
```

Commodities Exchange has several support library functions used to parse arguments:

```
tooltypearray = UBYTE **ArgArrayInit(LONG argc, UBYTE **argv);
                void    ArgArrayDone(void);

tooltypevalue = STRPTR  ArgString(UBYTE **tooltypearray, STRPTR tooltype, STRPTR defaultvalue);
tooltypevalue = LONG   *ArgInt(UBYTE **tooltypearray, STRPTR tooltype, LONG defaultvalue);
```

ArgArrayInit() initializes a Tool Type array of strings which it creates from the startup arguments, **argc** and **argv**. It doesn't matter if these startup arguments come from the Workbench or from a Shell, **ArgArrayInit()** can extract arguments from either source. Because **ArgArrayInit()** uses some icon.library functions, a commodity is responsible for opening that library before using the function.

ArgArrayInit() also uses some resources that must be returned to the system when the commodity is done. **ArgArrayDone()** performs this clean up. Like **ArgArrayInit()**, **ArgArrayDone()** uses icon.library, so the library has to remain open until **ArgArrayDone()** is finished.

The support library has two functions that use the Tool Type array set up by **ArgArrayInit()**, **ArgString()** and **ArgInt()**. **ArgString()** scans the Tool Type array for a specific Tool Type. If successful, it returns a pointer to the value associated with that Tool Type. If it doesn't find the Tool Type, it returns the default value passed to it. **ArgInt()** is similar to **ArgString()**. It also scans the **ArgArrayInit()**'s Tool Type array, but it returns a LONG rather than a string pointer. **ArgInt()** extracts the integer value associated with a Tool Type, or, if that Tool Type is not present, it returns the default value.

Of course, these Tool Type parsing functions are not restricted to the standard Commodities Exchange Tool Types. A commodity that requires any arguments should use these functions along with custom Tool Types to obtain these values. Because the Commodities Exchange standard arguments are processed as Tool Types, the user will expect to enter other arguments as Tool Types too.

Filter Objects and Input Description Strings

Because not all commodities are interested in every input event that makes it way down the input chain, Commodities Exchange has a method for filtering them. A filter **CxObject** compares the **CxMessages** it receives to a pattern. If a **CxMessage** matches the pattern, the filter diverts the **CxMessage** down its personal list of **CxObjects**.

```
CxObj *CxFilter(UBYTE *descriptionstring);
```

The C macro **CxFilter()** (defined in *<libraries/commodities.h>*) returns a pointer to a filter **CxObject**. The macro has only one argument, a pointer to a string describing which input events to filter. The following regular expression outlines the format of the input event description string (CX_POPKEY uses the same description string format):

[class] { [-] (qualifier | synonym)) } [[-] upstroke] [highmap |ANSICode]

Class can be any one of the class strings in the table below. Each class string corresponds to a class of input event as defined in *<devices/inputevent.h>*. Commodities Exchange will assume the class is rawkey if the class is not explicitly stated.

Class String	Input Event Class
"rawkey"	IECLASS_RAWKEY
"rawmouse"	IECLASS_RAWMOUSE
"event"	IECLASS_EVENT
"pointerpos"	IECLASS_POINTERPOS
"timer"	IECLASS_TIMER
"newprefs"	IECLASS_NEWPREFS
"diskremoved"	IECLASS_DISKREMOVED
"diskinserted"	IECLASS_DISKINSERTED

Qualifier is one of the qualifier strings from the table below. Each string corresponds to an input event qualifier as defined in *<devices/inputevent.h>*). A dash preceding the qualifier string tells the filter object not to care if that qualifier is present in the input event. Notice that there can be more than one qualifier (or none at all) in the input description string.

Qualifier String	Input Event Class
"lshift"	IEQUALIFIER_LSHIFT
"rshift"	IEQUALIFIER_RSHIFT
"capslock"	IEQUALIFIER_CAPSLOCK
"control"	IEQUALIFIER_CONTROL
"lalt"	IEQUALIFIER_LALT
"ralt"	IEQUALIFIER_RALT
"lcommand"	IEQUALIFIER_LCOMMAND
"rcommand"	IEQUALIFIER_RCOMMAND
"numericpad"	IEQUALIFIER_NUMERICPAD
"repeat"	IEQUALIFIER_REPEAT
"midbutton"	IEQUALIFIER_MIDBUTTON
"rbutton"	IEQUALIFIER_RBUTTON
"leftbutton"	IEQUALIFIER_LEFTBUTTON
"relativemouse"	IEQUALIFIER_RELATIVEMOUSE

Synonym is one of the synonym strings from the table below. These strings act as synonyms for groups of qualifiers. Each string corresponds to a synonym identifier as defined in *<libraries/commodities.h>*. A dash preceding the synonym string tells the filter object not to care if that synonym is present in the input event. Notice that there can be more than one synonym (or none at all) in the input description string.

Synonym String	Synonym Identifier	
"shift"	IXSYM_SHIFT	/* look for either shift key */
"caps"	IXSYM_CAPS	/* look for either shift key or capslock */
"alt"	IXSYM_ALT	/* look for either alt key */

Upstroke is the literal string "upstroke". If this string is absent, the filter considers only downstrokes. If it is present alone, the filter considers only upstrokes. If preceded by a dash, the filter considers both upstrokes and downstrokes.

Highmap is one of the following strings:

```
"space", "backspace", "tab", "enter", "return", "esc", "del", "up", "down", "right",
"left", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "f8", "f9", "f10", "help".
```

ANSICode is a single character (for example 'a') that Commodities Exchange looks up in the system default keymap.

Here are some example description strings. For function key F2 with the left Shift and either Alt key pressed, the input description string would be:

```
''rawkey lshift alt f2''
```

To specify the key that produces an 'a' (this may or may not be the A key depending on the keymap), with or without any Shift, Alt, or control keys pressed use:

```
''-shift -alt -control a''
```

For a mouse move with the right mouse button down, use:

```
''rawmouse rbutton''
```

To specify a timer event use:

```
''timer''
```

Connecting CxObjects

A **CxObject** has to be inserted into the Commodities network before it can process any **CxMessages**. **AttachCxObj()** adds a **CxObject** to the personal list of another **CxObject**. The HotKey.c example uses it to attach its filter to a broker.

```
void AttachCxObj ( CxObj *headobj, CxObj *co);
void InsertCxObj ( CxObj *headobj, CxObj *co, CxObj *co_pred );
void EnqueueCxObj( CxObj *headobj, CxObj *co );
void SetCxObjPri ( CxObj *co, long pri );
void RemoveCxObj ( CxObj *co );
```

AttachCxObj() adds the **CxObject** to the end of **headobj**'s personal list. The ordering of a **CxObject** list determines which object gets **CxMessages** first. **InsertCxObj()** also inserts a **CxObject**, but it inserts it after another **CxObject** already in the personal list (**co_pred** in the prototype above).

Brokers aren't the only **CxObjects** with a priority. All **CxObjects** have a priority associated with them. To change the priority of any **CxObject**, use the **SetCxObjPri()** function. A commodity can use the priority to keep **CxObjects** in a personal list sorted by their priority. The commodities.library function **EnqueueCxObj()** inserts a **CxObject** into another **CxObject**'s personal list based on priority.

Like its name implies, the **RemoveCxObj()** function removes a **CxObject** from a personal list. Note that it is not necessary to remove a **CxObject** from a list in order to delete it.

```
;/* HotKey.c - Simple hot key commodity compiled with SASC 5.10
LC -b0 -cfist -v -j73 hotkey.c
Blink FROM LIB:c.o,hotkey.o TO hotkey LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
 */
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/commodities_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

#define EVT_HOTKEY 1L

void main(int, char **);
void ProcessMsg(void);

struct Library *CxBase, *IconBase;
struct MsgPort *broker_mp;
CxObj *broker, *filter, *sender, *translate;

struct NewBroker newbroker = {
    NB_VERSION,
    "RKM HotKey",           /* string to identify this broker */
    "A Simple HotKey",
    "A simple hot key commodity",
    NBU_UNIQUE | NBU_NOTIFY,    /* Don't want any new commodities starting with this name. */
    0, 0, 0, 0                  /* If someone tries it, let me know */
};

ULONG cxsigflag;

void main(int argc, char **argv)
{
    UBYTE *hotkey, **ttypes;
    CxMsg *msg;

    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        /* open the icon.library for the support library */
        /* functions, ArgArrayInit() and ArgArrayDone()  */
        if (IconBase = OpenLibrary("icon.library", 36L))
        {
            if (broker_mp = CreateMsgPort())
            {
                newbroker.nb_Port = broker_mp;
                cxsigflag = 1L << broker_mp->mp_SigBit;

                /* ArgArrayInit() is a support library function (from the 2.0 version
                 * of amiga.lib) that makes it easy to read arguments from either a
                 * CLI or from Workbench's ToolTypes.  Because it uses icon.library,
                 * the library has to be open before calling this function.
                 * ArgArrayDone() cleans up after this function.
                 */
                ttypes = ArgArrayInit(argc, argv);

                /* ArgInt() (also from amiga.lib) searches through the array set up
                 * by ArgArrayInit() for a specific ToolType.  If it finds one, it
```

```
         * returns the numeric value of the number that followed the
         * ToolType (i.e., CX_PRIORITY=7). If it doesn't find the ToolType,
         * it returns the default value (the third argument)
         */
        newbroker.nb_Pri = (BYTE)ArgInt(ttypes, "CX_PRIORITY", 0);

        /* ArgString() works just like ArgInt(), except it returns a pointer to a string
         * rather than an integer. In the example below, if there is no ToolType
         * "HOTKEY", the function returns a pointer to "rawkey control esc".
         */
        hotkey = ArgString(ttypes, "HOTKEY", "rawkey control esc");

        if (broker = CxBroker(&newbroker, NULL))
        {
            /* CxFilter() is a macro that creates a filter CxObject.  This filter
             * passes input events that match the string pointed to by hotkey.
             */
            if (filter = CxFilter(hotkey))
            {
                /* Add a CxObject to another's personal list */
                AttachCxObj(broker, filter);

                /* CxSender() creates a sender CxObject.  Every time a sender gets
                 * a CxMessage, it sends a new CxMessage to the port pointed to in
                 * the first argument. CxSender()'s second argument will be the ID
                 * of any CxMessages the sender sends to the port.  The data pointer
                 * associated with the CxMessage will point to a *COPY* of the
                 * InputEvent structure associated with the orginal CxMessage.
                 */
                if (sender = CxSender(broker_mp, EVT_HOTKEY))
                {
                    AttachCxObj(filter, sender);

                    /* CxTranslate() creates a translate CxObject. When a translate
                     * CxObject gets a CxMessage, it deletes the original CxMessage
                     * and adds a new input event to the input.device's input stream
                     * after the Commodities input handler. CxTranslate's argument
                     * points to an InputEvent structure from which to create the new
                     * input event.  In this example, the pointer is NULL, meaning no
                     * new event should be introduced, which causes any event that
                     * reaches this object to disappear from the input stream.
                     */
                    if (translate = (CxTranslate(NULL)))
                    {
                        AttachCxObj(filter, translate);

                        /* CxObjError() is a commodities.library function that returns
                         * the internal accumulated error code of a CxObject.
                         */
                        if (! CxObjError(filter))
                        {
                            ActivateCxObj(broker, 1L);
                            ProcessMsg();
                        }
                    }
                }
            }
            /* DeleteCxObjAll() is a commodities.library function that not only
             * deletes the CxObject pointed to in its argument, but it deletes
             * all of the CxObjects that are attached to it.
             */
            DeleteCxObjAll(broker);

            /* Empty the port of all CxMsgs */
            while(msg = (CxMsg *)GetMsg(broker_mp))
                ReplyMsg((struct Message *)msg);
        }
        DeletePort(broker_mp);
    }
    /* this amiga.lib function cleans up after ArgArrayInit() */
    ArgArrayDone();
    CloseLibrary(IconBase);
  }
  CloseLibrary(CxBase);
 }
}
```

```
void ProcessMsg(void)
{
    extern struct MsgPort *broker_mp;
    extern CxObj *broker;
    extern ULONG cxsigflag;
    CxMsg *msg;
    ULONG sigrcvd, msgid, msgtype;
    LONG returnvalue = 1L;

    while(returnvalue)
    {
        sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

        while(msg = (CxMsg *)GetMsg(broker_mp))
        {
            msgid = CxMsgID(msg);
            msgtype = CxMsgType(msg);
            ReplyMsg((struct Message *)msg);

            switch(msgtype)
            {
                case CXM_IEVENT:
                    printf("A CXM_EVENT, ");
                    switch(msgid)
                    {
                        case EVT_HOTKEY: /* We got the message from the sender CxObject */
                            printf("You hit the HotKey.\n");
                            break;
                        default:
                            printf("unknown.\n");
                            break;
                    }
                    break;
                case CXM_COMMAND:
                    printf("A command: ");
                    switch(msgid)
                    {
                        case CXCMD_DISABLE:
                            printf("CXCMD_DISABLE\n");
                            ActivateCxObj(broker, 0L);
                            break;
                        case CXCMD_ENABLE:
                            printf("CXCMD_ENABLE\n");
                            ActivateCxObj(broker, 1L);
                            break;
                        case CXCMD_KILL:
                            printf("CXCMD_KILL\n");
                            returnvalue = 0L;
                            break;
                        case CXCMD_UNIQUE:
                        /* Commodities Exchange can be told not only to refuse to launch a
                         * commodity with a name already in use but also can notify the
                         * already running commodity that it happened. It does this by
                         * sending a CXM_COMMAND with the ID set to CXMCMD_UNIQUE.  If the
                         * user tries to run a windowless commodity that is already running,
                         * the user wants the commodity to shut down. */
                            printf("CXCMD_UNIQUE\n");
                            returnvalue = 0L;
                            break;
                        default:
                            printf("Unknown msgid\n");
                            break;
                    }
                    break;
                default:
                    printf("Unknown msgtype\n");
                    break;
            }
        }
        if (sigrcvd & SIGBREAKF_CTRL_C)
        {
            returnvalue = 0L;
            printf("CTRL C signal break\n");
        }
    }
}
```

Sender CxObjects

A filter **CxObject** by itself is not especially useful. It needs some other **CxObjects** attached to it. A commodity interested in knowing if a specific key was pressed uses a filter to detect and divert the corresponding **CxMessage** down the filter's personal list. The filter does this without letting the commodity know what happened. The sender **CxObject** can be attached to a filter to notify a commodity that it received a **CxMessage**. **CxSender()** is a macro that creates a sender **CxObject**.

```
senderCxObj = CxObj *CxSender(struct MsgPort *senderport, LONG cxmID);
```

CxSender() supplies the sender with an Exec message port and an ID. For every **CxMessage** a sender receives, it sends a new **CxMessage** to the Exec message port passed in **CxSender()**. Normally, the commodity creates this port. It is not unusual for a commodity's broker and sender(s) to share an Exec message port. The HotKey.c example does this to avoid creating unnecessary message ports. A sender uses the ID (**cxmID**) passed to **CxSender()** as the ID for all the **CxMessages** that the it transmits. A commodity uses the ID to monitor **CxMessages** from several senders at a single message port.

A sender does several things when it receives a **CxMessage**. First, it duplicates the **CxMessage**'s corresponding input event and creates a new **CxMessage**. Then, it points the new **CxMessage**'s data field to the copy of the input event and sets the new **CxMessage**'s ID to the ID passed to **CxSender()**. Finally, it sends the new **CxMessage** to the port passed to **CxSender()**, asynchronously.

Because HotKey uses only one message port between its broker and sender object, it has to extract the **CxMessage**'s type so it can tell if it is a CXM_IEVENT or a CXM_COMMAND. If HotKey gets a CXM_IEVENT, it compares the **CxMessage**'s ID to the sender's ID, EVT_HOTKEY, to see which sender sent the **CxMessage**. Of course HotKey has only one sender, so it only checks for only one ID. If it had more senders, HotKey would check for the ID of each of the other senders as well.

Although HotKey doesn't use it, a CXM_IEVENT **CxMessage** contains a pointer to the copy of an input event. A commodity can extract this pointer (using **CxMsgData()**) if it needs to examine the input event copy. This pointer is only valid before the **CxMessage** reply. Note that it does not make any sense to modify the input event copy.

Senders are attached almost exclusively to **CxObjects** that filter out most input events (usually a filter **CxObject**). Because a sender sends a **CxMessage** for every single input event it gets, it should only get a select few input events. The **AttachCxObj()** function can add a **CxObject** to the end of a filter's (or some other filtering **CxObject's**) personal list. A commodity should not attach a **CxObject** to a sender as a sender ignores any **CxObjects** in its personal list.

Translate CxObjects

Normally, after a commodity processes a hot key input event, it needs to eliminate that input event. Other commodities may need to replace an input event with a different one. The translate **CxObject** can be used for these purposes.

```
translateCxObj = CxObj  *CxTranslate(struct InputEvent *newinputevent);
```

The macro **CxTranslate()** creates a new translate **CxObject**. **CxTranslate()**'s only argument is a pointer to a chain of one or more **InputEvent** structures.

When a translate **CxObject** receives a **CxMessage**, it eliminates the **CxMessage** and its corresponding input event from the system. The translator introduces a new input event, which Commodities Exchange copies from the **InputEvent** structure passed to **CxTranslate()** (newinputevent from the function prototype above), in place of the deleted input event.

A translator is normally attached to some kind of filtering **CxObject**. If it wasn't, it would translate all input events into the same exact input event. Like the sender **CxObject**, a translator does not divert **CxMessages** down its personal list, so it doesn't serve any purpose to add any to it.

```
void SetTranslate( CxObj *translator, struct InputEvent *ie );
```

It is possible to change the **InputEvent** structure that a translator looks at when it creates and introduces new input events into the input stream. The function **SetTranslate()** accepts a pointer to the new **InputEvent** structure, which the translator will duplicate and introduce when it receives a **CxMessage**.

HotKey utilizes a special kind of translator. Instead of supplying a new input event, HotKey passes a NULL to **CxTranslate()**. If a translator has a NULL new input event pointer, it does not introduce a new input event, but still eliminates any **CxMessages** and corresponding input events it receives.

CxObject Errors

A Commodities Exchange function that acts on a **CxObject** records errors in the **CxObject**'s accumulated error field. The function **CxObjError()** returns a **CxObject**'s error field.

```
co_errorfield = LONG CxObjError( CxObj *co );
```

Each bit in the error field corresponds to a specific type of error. The following is a list of the currently defined **CxObject** errors and their corresponding bit mask constants.

Error Constant	Meaning
COERR_ISNULL	**CxObjError()** was passed a NULL.
COERR_NULLATTACH	Someone tried to attach a NULL **CxObject** to this **CxObject**.
COERR_BADFILTER	This filter **CxObject** currently has an invalid filter description.
COERR_BADTYPE	Someone tried to perform a type specific function on the wrong type of **CxObject** (for example calling **SetFilter()** on a sender **CxObject**).

The remaining bits are reserved by Commodore for future use. HotKey.c checks the error field of its filter **CxObject** to make sure the filter is valid. HotKey.c does not need to check the other objects with **CxObjError()** because it already makes sure that these other objects are not NULL, which is the only other kind of error the other objects can cause in this situation.

Commodities Exchange has a function that clears a **CxObject**'s accumulated error field, **ClearCxObjError()**.

```
void ClearCxObjError( CxObj *co );
```

A commodity should be careful about using this, especially on a filter. If a commodity clears a filter's error field and the COERR_BADFILTER bit is set, Commodities Exchange will think that the filter is OK and start sending messages through it.

Uniqueness

When a commodity opens its broker, it can ask Commodities Exchange not to launch another broker with the same name (**nb_Name**). The purpose of the uniqueness feature is to prevent the user from starting duplicate commodities. If a commodity asks, Commodities Exchange will not only refuse to create a new, similarly named broker, but it will also notify the original commodity if someone tries to do so.

A commodity tells Commodities Exchange not to allow duplicates by setting certain bits in the **nb_Unique** field of the **NewBroker** structure it sends to **CxBroker()**:

```
NBU_UNIQUE      bit 0
NBU_NOTIFY      bit 1
```

Setting the NBU_UNIQUE bit prevents duplicate commodities. Setting the NBU_NOTIFY bit tells Commodities Exchange to notify a commodity if an attempt was made to launch a duplicate. Such a commodity will receive a CXM_COMMAND **CxMessage** with an ID of CXCMD_UNIQUE when someone tries to duplicate it. Because the uniqueness feature uses the name a programmer gives a commodity to differentiate it from other commodities, it is possible for completely different commodities to share the same name, preventing the two from coexisting. For this reason, a commodity should not use a name that is likely to be in use by other commodities (like ''filter'' or ''hotkey''). Instead, use a name that matches the commodity name.

When HotKey.c gets a CXCMD_UNIQUE **CxMessage**, it shuts itself down. HotKey.c and all the windowless commodities that come with the Release 2 Workbench shut themselves down when they get a CXCMD_UNIQUE **CxMessage**. Because the user will expect all windowless commodities to work this way, all windowless commodities should follow this standard.

When the user tries to launch a duplicate of a system commodity that has a window, the system commodity moves its window to the front of the display, as if the user had clicked the ''Show'' gadget in the controller program's window. A windowed commodity should mimic conventions set by existing windowed system commodities, and move its window to the front of the display.

Signal CxObjects

A commodity can use a sender **CxObject** to find out if a **CxMessage** has ''visited'' a **CxObject**, but this method unnecessarily uses system resources. A commodity that is only interested in knowing if such a visitation took place does not need to see a corresponding input event or a **CxMessage** ID. Instead, Commodities Exchange has a **CxObject** that uses an Exec signal.

```
signalCxObj = CxObj  *CxSignal(struct Task *, LONG cx_signal);
```

CxSignal() sets up a signal **CxObject**. When a signal **CxObject** receives a **CxMessage**, it signals a task. The commodity is responsible for determining the proper task ID and allocating the signal. Normally, a commodity wants to be signalled so it uses **FindTask**(NULL) to find it's own task address. Note that **cx_signal** from the above prototype is the signal number as returned by **AllocSignal()**, not the signal mask made from that number. For more information on signals, see the Exec chapter.

The example Divert.c (shown a little later in this chapter) uses a signal **CxObject**.

Custom CxObjects

Although the **CxObjects** mentioned so far take care of most of the input event handling a commodity needs to do, they cannot do it all. This is why Commodities Exchange has a custom **CxObject**. When a custom **CxObject** receives a **CxMessage**, it calls a function provided by the commodity.

```
customCxObj = CxObj  *CxCustom(LONG *customfunction(), LONG  cxmID);
```

A custom **CxObject** is the only means by which a commodity can directly modify input events as they pass through the Commodities network as **CxMessages**. For this reason, it is probably the most dangerous of the **CxObjects** to use.

> *A Warning About Custom CxObjects.* Unlike the rest of the code a commodities programmer writes, the code passed to a custom **CxObject** runs as part of the input.device task, putting severe restrictions on the function. No DOS or Intuition functions can be called. No assumptions can be made about the values of registers upon entry. Any function passed to **CxCustom()** should be very quick and very simple, with a minimum of stack usage.

Commodities Exchange calls a custom **CxObject**'s function as follows:

```
void customfunction(CxMsg *cxm, CxObj *customcxobj);
```

where **cxm** is a pointer to a **CxMessage** corresponding to a real input event, and **customcxobj** is a pointer to the custom **CxObject**. The custom function can extract the pointer to the input event by calling **CxMsgData()**. Before passing the **CxMessage** to the custom function, Commodities Exchange sets the **CxMessage**'s ID to the ID passed to **CxCustom()**.

The following is an example of a custom **CxObject** function that swaps the function of the left and right mouse buttons.

```
custom = CxCustom(CxFunction, 0L)

/* The custom function for the custom CxObject.  Any code for a custom CxObj must be */
/* short and sweet. This code runs as part of the input.device task */
#define CODEMASK (0x00FF & IECODE_LBUTTON & IECODE_RBUTTON)
void CxFunction(register CxMsg *cxm, CxObj *co)
{
    struct InputEvent *ie;
    UWORD mousequals = 0x0000;

    /* Get the struct InputEvent associated with this CxMsg.  Unlike the InputEvent
     * extracted from a CxSender's CxMsg, this is a *REAL* input event, be careful with it.
     */
    ie = (struct InputEvent *)CxMsgData(cxm);

    /* Check to see if this input event is a left or right mouse button   */
    /* by itself (a mouse button can also be a qualifier).  If it is, flip the   */
    /* low order bit to switch leftbutton <--> rightbutton. */
    if (ie->ie_Class == IECLASS_RAWMOUSE)
        if ((ie->ie_Code & CODEMASK) == CODEMASK)  ie->ie_Code ^= 0x0001;

    /* Check the qualifiers. If a mouse button was down when this */
    /* input event occurred, set the other mouse button bit.      */
    if (ie->ie_Qualifier & IEQUALIFIER_RBUTTON)  mousequals |= IEQUALIFIER_LEFTBUTTON;
    if (ie->ie_Qualifier & IEQUALIFIER_LEFTBUTTON)  mousequals |= IEQUALIFIER_RBUTTON;

    /* clear the RBUTTON and LEFTBUTTON qualifier bits */
    ie->ie_Qualifier &= ~(IEQUALIFIER_LEFTBUTTON | IEQUALIFIER_RBUTTON);

    /* set the mouse button qualifier bits to their new values */
    ie->ie_Qualifier |= mousequals;
}
```

Debug CxObjects

The final **CxObject** is the debug **CxObject**. When a debug **CxObject** receives a **CxMessage**, it sends debugging information to the serial port using **kprintf()**.

```
debugCxObj  = CxObj  *CxDebug(LONG ID);
```

The debug **CxObject** will **kprintf()** the following information about itself, the **CxMsg**, and the corresponding **InputEvent** structure:

```
DEBUG NODE: 7CB5AB0, ID: 2
 CxMsg: 7CA6EF2, type: 0, data 2007CA destination 6F1E07CB
dump IE: 7CA6F1E
 Class 1
 Code 40
 Qualifier 8000
 EventAddress 40001802
```

There has to be a terminal connected to the Amiga's serial port to receive this information. See the **kprintf()** Autodoc (debug.lib) for more details. Note that the debug **CxObject** did not work before V37.

The IX Structure

Commodities Exchange does not use the input event description strings discussed earlier to match input events. Instead, Commodities Exchange converts these strings to its own internal format. These input expressions are available for commodities to use instead of the input description strings. The following is the **IX** structure as defined in *<libraries/commodities.h>*:

```
#define IX_VERSION   2

struct InputXpression {
   UBYTE    ix_Version;      /* must be set to IX_VERSION  */
   UBYTE    ix_Class;        /* class must match exactly   */
   UWORD    ix_Code;
   UWORD    ix_CodeMask;     /* normally used for UPCODE   */
   UWORD    ix_Qualifier;
   UWORD    ix_QualMask;
   UWORD    ix_QualSame;     /* synonyms in qualifier      */
   };
typedef struct InputXpression IX;
```

The **ix_Version** field contains the current version number of the **InputXpression** structure. The current version is defined as IX_VERSION. The **ix_Class** field contains the IECLASS_ constant (defined in *<devices/inputevent.h>*) of the class of input event sought. Commodities Exchange uses the **ix_Code** and **ix_CodeMask** fields to match the **ie_Code** field of a struct **InputEvent**. The bits of **ix_CodeMask** indicate which bits are relevant in the **ix_Code** field when trying to match against a **ie_Code**. If any bits in **ix_CodeMask** are off, Commodities Exchange does not consider the corresponding bit in **ie_Code** when trying to match input events. This is used primarily to mask out the IECODE_UP_PREFIX bit of rawkey events, making it easier to match both up and down presses of a particular key.

IX's qualifier fields, **ix_Qualifier**, **ix_QualMask**, and **ix_QualSame**, are used to match the **ie_Qualifier** field of an **InputEvent** structure. The **ix_Qualifier** and **ix_QualMask** fields work just like **ix_Code** and **ix_CodeMask.** The bits of **ix_QualMask** indicate which bits are relevant when comparing **ix_Qualifier** to **ie_Qualifier**. The **ix_QualSame** field tells Commodities Exchange that certain qualifiers are equivalent:

```
#define IXSYM_SHIFT  1     /* left- and right- shift are equivalent      */
#define IXSYM_CAPS   2     /* either shift or caps lock are equivalent  */
#define IXSYM_ALT    4     /* left- and right- alt are equivalent        */
```

For example, the input description string

```
"rawkey -caps -lalt -relativemouse -upstroke ralt tab"
```

matches a tab upstroke or downstroke with the right Alt key pressed whether or not the left Alt, either Shift, or the Caps Lock keys are down. The following **IX** structure corresponds to that input description string:

```
IX ix = {
    IX_VERSION,                    /* The version */
    IECLASS_RAWKEY,                /* We're looking for a RAWKEY event */
    0x42,                          /* The key the usa0 keymap maps to a tab*/
    0x00FF & (~IECODE_UP_PREFIX), /* We want up and down key presses */
    IEQUALIFIER_RALT,              /* The right alt key must be down */
    0xFFFF & ~(IEQUALIFIER_LALT | IEQUALIFIER_LSHIFT |
        IEQUALIFIER_RSHIFT | IEQUALIFIER_CAPSLOCK | IEQUALIFIER_RELATIVEMOUSE),
        /* don't care about left alt, shift, capslock, or relativemouse qualifiers   */
    IXSYM_CAPS  /* The shift keys and the capslock key qualifiers are all equivalent */
};
```

The **CxFilter()** macro only accepts a description string to describe an input event. A commodity can change this filter, however, with the **SetFilter()** and **SetFilterIX()** function calls.

```
void SetFilter( CxObj *filter, UBYTE *descrstring );
void SetFilterIX( CxObj *filter, IX *ix );
```

SetFilter() and **SetFilterIX()** change which input events a filter **CxObject** diverts. **SetFilter()** accepts a pointer to an input description string. **SetFilterIX()** accepts a pointer to an **IX** input expression. A commodity that uses either of these functions should check the filter's error code with **CxObjError()** to make sure the change worked.

The function **ParseIX()** parses an input description string and translates it into an **IX** input expression.

```
errorcode = LONG ParseIX( UBYTE *descrstring, IX *ix );
```

Commodities Exchange uses **ParseIX()** to convert the description string in **CxFilter()** to an **IX** input expression. As was mentioned previously, as of commodities.library version 37.3, **ParseIX()** does not work with certain kinds of input strings.

Controlling CxMessages

A Custom **CxObject** has the power to directly manipulate the **CxMessages** that travel around the Commodities network. One way is to directly change values in the corresponding input event. Another way is to redirect (or dispose of) the **CxMessages**.

```
void DivertCxMsg  ( CxMsg *cxm, CxObj *headobj, CxObj *retobj );
void RouteCxMsg   ( CxMsg *cxm, CxObj *co );
void DisposeCxMsg( CxMsg *cxm );
```

DivertCxMsg() and **RouteCxMsg()** dictate where the **CxMessage** will go next. Conceptually, **DivertCxMsg()** is analogous to a subroutine in a program; the **CxMessage** will travel down the personal list of a **CxObject** (**headobj** in the prototype) until it gets to the end of that list. It then returns and visits the **CxObject** that follows the return **CxObject** (the return **CxObject** in the prototype above is **retobj**). **RouteCxMsg()** is analogous to a goto in a program; it has no **CxObject** to return to.

DisposeCxMsg() removes a **CxMessage** from the network and releases its resources. The translate **CxObject** uses this function to remove a **CxMessage**.

The example Divert.c shows how to use **DivertCxMsg()** as well as a signal **CxObject**.

```
;/* divert.c - commodity to monitor user inactivity - compiled with SASC 5.10
LC -b0 -cfist -v -j73 divert.c
Blink FROM LIB:c.o,divert.o TO divert LIBRARY LIB:LC.lib,LIB:Amiga.lib NODEBUG SC SD
quit; */
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>
#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/commodities_protos.h>
#include <devices/inputevent.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

#define TIMER_CLICKS 100

void main(int, char **);
void ProcessMsg(void);
void CxFunction(CxMsg *, CxObj *);

struct Library *CxBase, *IconBase;
struct MsgPort *broker_mp;
CxObj *broker, *cocustom, *cosignal;

struct NewBroker newbroker =
{
    NB_VERSION,
    "Divert",           /* string to identify this broker */
    "Divert",
    "show divert",
    NBU_UNIQUE | NBU_NOTIFY,  /* Don't want any new commodities starting with this name. */
    0, 0, 0, 0                /* If someone tries it, let me know                         */
};

struct Task *task;
ULONG cxsigflag, signal, cxobjsignal;

void main(int argc, char **argv)
{
    UBYTE **ttypes;
    CxMsg *msg;

    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        /* open the icon.library for support library functions, ArgArrayInit() and ArgArrayDone() */
        if (IconBase = OpenLibrary("icon.library", 36L))
        {
            if (broker_mp = CreateMsgPort())
            {
                newbroker.nb_Port = broker_mp;
                cxsigflag = 1L << broker_mp->mp_SigBit;

                /* ArgArrayInit() is a support library function (in the 2.0 version of amiga.lib) */
                /* that makes it easy to read arguments from either a CLI or from Workbench's     */
                /* ToolTypes. Because it uses icon.library, the library has to be open before     */
                /* before calling this function.  ArgArrayDone() cleans up after this function.   */
                ttypes = ArgArrayInit(argc, argv);

                /* ArgInt() (in amiga.lib) searches through the array set up by ArgArrayInit()    */
                /* for a specific ToolType.  If it finds one, it returns the numeric value of the */
                /* number that followed the ToolType (i.e., CX_PRIORITY=7).  If it  doesn't find  */
                /* the ToolType, it returns the default value (the third argument)                */
                newbroker.nb_Pri = (BYTE)ArgInt(ttypes, "CX_PRIORITY", 0);

                if (broker = CxBroker(&newbroker, NULL))
                {
                    /* CxCustom() takes two arguments, a pointer to the custom function          */
                    /* and an ID. Commodities Exchange will assign that ID to any CxMsg          */
                    /* passed to the custom  function.                                           */
```

```
                    if (cocustom = CxCustom(CxFunction, 0L))
                    {
                        AttachCxObj(broker, cocustom);

                        /* Allocate a signal bit for the signal CxObj */
                        if ( (signal = (ULONG)AllocSignal(-1L)) != -1)
                        {
                            /* set up the signal mask */
                            cxobjsignal = 1L << signal;
                            cxsigflag |= cxobjsignal;

                            /* CxSignal takes two arguments, a pointer to the task to signal      */
                            /* (normally the commodity) and the number of the signal bit the      */
                            /* commodity acquired to signal with.                                 */
                            task = FindTask(NULL);
                            if (cosignal = CxSignal(task, signal))
                            {
                                AttachCxObj(cocustom, cosignal);
                                ActivateCxObj(broker, 1L);
                                ProcessMsg();
                            }
                            FreeSignal(signal);
                        }
                    }
                    /* DeleteCxObjAll() is a commodities.library function that not only deletes   */
                    /* the CxObject pointed to in its argument, but it deletes all of the         */
                    /* CxObjects that are attached to it.                                         */
                    DeleteCxObjAll(broker);

                    /* Empty the port of all CxMsgs */
                    while(msg = (CxMsg *)GetMsg(broker_mp))
                        ReplyMsg((struct Message *)msg);
                }
                DeletePort(broker_mp);
            }
            ArgArrayDone();   /* this amiga.lib function cleans up after ArgArrayInit()           */
            CloseLibrary(IconBase);
        }
        CloseLibrary(CxBase);
    }
}

void ProcessMsg(void)
{
    extern struct MsgPort *broker_mp;
    extern CxObj *broker;
    extern ULONG cxsigflag;
    CxMsg *msg;
    ULONG sigrcvd, msgid;
    LONG returnvalue = 1L;

    while (returnvalue)
    {
        sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

        while(msg = (CxMsg *)GetMsg(broker_mp))
        {
            msgid = CxMsgID(msg);
            ReplyMsg((struct Message *)msg);

            switch(msgid)
            {
                case CXCMD_DISABLE:
                    ActivateCxObj(broker, 0L);
                    break;
                case CXCMD_ENABLE:
                    ActivateCxObj(broker, 1L);
                    break;
                case CXCMD_KILL:
                    returnvalue = 0L;
                    break;
                case CXCMD_UNIQUE:
                    returnvalue = 0L;
                    break;
            }
        }
```

```
        if (sigrcvd & SIGBREAKF_CTRL_C) returnvalue = 0L;

        /* Check to see if the signal CxObj signalled us. */
        if (sigrcvd & cxobjsignal) printf("Got Signal\n");
    }
}

/* The custom function for the custom CxObject.  Any code for a custom CxObj must be short       */
/* and sweet because it runs as part of the input.device task.                                  */
void CxFunction(register CxMsg *cxm, CxObj *co)
{
    struct InputEvent *ie;
    static ULONG time = 0L;

    /* Get the struct InputEvent associated with this CxMsg. Unlike the InputEvent              */
    /* extracted from a CxSender's CxMsg, this is a *REAL* input event, be careful with it.     */
    ie = (struct InputEvent *)CxMsgData(cxm);

    /* This custom function counts the number of timer events that go by while no other input   */
    /* events occur.  If it counts more than a certain amount of timer events, it clears the    */
    /* count and diverts the timer event CxMsg to the custom object's personal                  */
    /* list.  If an event besides a timer event passes by, the timer event count is reset.     */
    if (ie->ie_Class == IECLASS_TIMER)
    {
        time++;
        if (time >= TIMER_CLICKS)
        {
            time = 0L;
            DivertCxMsg(cxm, co, co);
        }
    }
    else
        time = 0L;
}
```

New Input Events

Commodities Exchange also has functions used to introduce new input events to the input stream.

```
struct InputEvent *InvertString( UBYTE *string, ULONG *keymap );
void               FreeIEvents( struct InputEvent *ie );
void               AddIEvents( struct InputEvent *ie );
```

InvertString() is an amiga.lib function that accepts an ASCII string and creates a linked list of input events that translate into the string using the supplied keymap (or the system default if the key map is NULL). The NULL terminated string may contain ANSI character codes, an input description enclosed in angle (<>) brackets, or one of the following backslash escape characters:

\r - return
\t - tab
\\ - backslash

For example:

abc<alt f1>\rhi there.

FreeIEvents() frees a list of input events allocated by **InvertString()**. **AddIEvents()** is a commodities.library function that adds a linked list of input events at the the top of the Commodities network. Each input event in the list is made into an individual **CxMessage**. Note that if passed a linked list of input events created by **InvertString()**, the order the events appear in the string will be reversed.

```
;/* PopShell.c - Simple hot key commodity compiled with SASC 5.10
LC -b0 -cfist -v -j73 popshell.c
Blink FROM LIB:c.o,popshell.o TO popshell LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
#include <exec/libraries.h>
#include <libraries/commodities.h>
#include <dos/dos.h>

#include <clib/exec_protos.h>
#include <clib/alib_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/commodities_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif


void main(int, char **);
void ProcessMsg(void);

#define EVT_HOTKEY 1L
struct Library *CxBase, *IconBase;
struct MsgPort *broker_mp;
CxObj *broker, *filter, *sender, *translate;

struct NewBroker newbroker =
{
    NB_VERSION,
    "RKM PopShell",           /* string to identify this broker */
    "A Simple PopShell",
    "A simple PopShell commodity",
    NBU_UNIQUE | NBU_NOTIFY,      /* Don't want any new commodities starting with this name. */
    0, 0, 0, 0                    /* If someone tries it, let me know */
};

UBYTE *newshell = "\rllehswen";  /* "newshell" spelled backwards */
struct InputEvent *ie;
ULONG cxsigflag;


void main(int argc, char **argv)
{
    UBYTE *hotkey, **ttypes;
    CxMsg *msg;

    if (CxBase = OpenLibrary("commodities.library", 37L))
    {
        if (IconBase = OpenLibrary("icon.library", 36L))
        {
            if (broker_mp = CreateMsgPort())
            {
                newbroker.nb_Port = broker_mp;
                cxsigflag = 1L << broker_mp->mp_SigBit;
                ttypes = ArgArrayInit(argc, argv);
                newbroker.nb_Pri = (BYTE)ArgInt(ttypes, "CX_PRIORITY", 0);
                hotkey = ArgString(ttypes, "HOTKEY", "rawkey control esc");

                if (broker = CxBroker(&newbroker, NULL))
                {
                    /* HotKey() is an amiga.lib function that creates a filter, sender */
                    /* and translate CxObject and connects them to report a hot key    */
                    /* press and delete its input event. */
                    if (filter = HotKey(hotkey, broker_mp, EVT_HOTKEY))
                    {
                        AttachCxObj(broker, filter); /* Add a CxObject to another's personal list */

                        if (! CxObjError(filter))
                        {
                            /* InvertString() is an amiga.lib function that creates a linked */
                            /* list of input events which would translate into the string    */
                            /* passed to it.  Note that it puts the input events in the      */
                            /* opposite order in which the corresponding letters appear in   */
                            /* the string.  A translate CxObject expects them backwards.     */
```

```
                    if (ie = InvertString(newshell, NULL))
                    {
                        ActivateCxObj(broker, 1L);
                        ProcessMsg();
                        /* we have to release the memory allocated by InvertString.       */
                        FreeIEvents(ie);
                    }
                }
            }
            /* DeleteCxObjAll() is a commodities.library function that not only      */
            /* deletes the CxObject pointed to in its argument, but deletes all of   */
            /* the CxObjects attached to it.                                          */
            DeleteCxObjAll(broker);

            /* Empty the port of all CxMsgs */
            while(msg = (CxMsg *)GetMsg(broker_mp))
                ReplyMsg((struct Message *)msg);
        }
        DeletePort(broker_mp);
    }
    ArgArrayDone();  /* this amiga.lib function cleans up after ArgArrayInit() */
    CloseLibrary(IconBase);
  }
  CloseLibrary(CxBase);
 }
}

void ProcessMsg(void)
{
    extern struct MsgPort *broker_mp;
    extern CxObj *broker;
    extern ULONG cxsigflag;
    CxMsg *msg;
    ULONG sigrcvd, msgid, msgtype;
    LONG returnvalue = 1L;

    while (returnvalue)
    {
        sigrcvd = Wait(SIGBREAKF_CTRL_C | cxsigflag);

        while(msg = (CxMsg *)GetMsg(broker_mp))
        {
            msgid = CxMsgID(msg);
            msgtype = CxMsgType(msg);
            ReplyMsg((struct Message *)msg);

            switch(msgtype)
            {
                case CXM_IEVENT:
                    printf("A CXM_EVENT, ");
                    switch(msgid)
                    {
                        case EVT_HOTKEY:
                            /* We got the message from the sender CxObject */
                            printf("You hit the HotKey.\n");
                            /* Add the string "newshell" to input * stream.  If a shell       */
                            /* gets it, it'll open a new shell.                                */
                            AddIEvents(ie);
                            break;
                        default:
                            printf("unknown.\n");
                            break;
                    }
                    break;
                case CXM_COMMAND:
                    printf("A command: ");
                    switch(msgid)
                    {
                        case CXCMD_DISABLE:
                            printf("CXCMD_DISABLE\n");
                            ActivateCxObj(broker, 0L);
                            break;
                        case CXCMD_ENABLE:
                            printf("CXCMD_ENABLE\n");
                            ActivateCxObj(broker, 1L);
                            break;
```

```
                    case CXCMD_KILL:
                        printf("CXCMD_KILL\n");
                        returnvalue = 0L;
                        break;
                    case CXCMD_UNIQUE:
                    /* Commodities Exchange can be told not only to refuse to launch a    */
                    /* commodity with a name already in use but also can notify the       */
                    /* already running commodity that it happened.  It does this by       */
                    /* sending a CXM_COMMAND with the ID set to CXMCMD_UNIQUE.  If the    */
                    /* user tries to run a windowless commodity that is already running,  */
                    /* the user wants the commodity to shut down.                         */
                        printf("CXCMD_UNIQUE\n");
                        returnvalue = 0L;
                        break;
                    default:
                        printf("Unknown msgid\n");
                        break;
                }
                break;
            default:
                printf("Unknown msgtype\n");
                break;
        }
    }

    if (sigrcvd & SIGBREAKF_CTRL_C)
    {
        returnvalue = 0L;
        printf("CTRL C signal break\n");
    }
  }
}
```

Function Reference

The following are brief descriptions of the Commodities Exchange functions covered in this chapter. All of these functions require Release 2 or a later version of the Amiga operating system. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 31-2: Commodities Exchange Functions

Function	Description
CxBroker()	Creates a **CxObject** of type Broker.
CxFilter()	Creates a **CxObject** of type Filter.
CxSender()	Creates a **CxObject** of type Sender.
CxTranslate()	Creates a **CxObject** of type Translate.
CxSignal()	Creates a **CxObject** of type Signal.
CxCustom()	Creates a **CxObject** of type Custom.
CxDebug()	Creates a **CxObject** of type Debug.
DeleteCxObj()	Frees a single **CxObject**
DeleteCxObjAll()	Frees a group of connected **CxObjects**
ActivateCxObj()	Activates a newly created **CxObject** in the commodities network.
CxTranslate()	Sets up substitution of one input event for another by translate **CxObjects**.
CxMsgType()	Finds the type of a **CxMessage**.
CxMsgData()	Returns the **CxMessage** data.
CxMsgID()	Returns the **CxMessage** ID.
CxObjError()	Returns the **CxObject**'s accumulated error field.
ClearCxObjError()	Clear the **CxObject**'s accumulated error field.
ArgArrayInit()	Create a Tool Types array from **argc** and **argv** (Workbench or Shell).
ArgArrayDone()	Free the resources used by **ArgArrayInit()**.
ArgString()	Return the string associated with a given Tool Type in the array.
ArgInt()	Return the integer associated with a given Tool Type in the array.
AttachCxObj()	Attaches a **CxObject** to the end of a given **CxObject**'s list.
InsertCxObj()	Inserts a **CxObject** in a given position in a **CxObject**'s list.
EnqueueCxObj()	Inserts a **CxObject** in a **CxObject**'s list by priority.
SetCxObjPri()	Sets a **CxObject**'s priority for **EnqueueCxObj()**.
RemoveCxObj()	Removes a **CxObject** from a list.
SetFilter()	Set a filter for a **CxObject** from an input description string.
SetFilterIX()	Set a filter for a **CxObject** from an **IX** data structure.
ParseIX()	Convert an input description string to an **IX** data structure.
DivertCxMsg()	Divert a **CxMessage** to one **CxObject** and return it to another.
RouteCxMsg()	Redirect a **CxMessage** to a new **CxObject**.
DisposeCxMsg()	Cancel a **CxMessage** removing it from the Commodities network.
InvertString()	Creates a linked list of input events that correspond to a given string.
FreeIEvents()	Frees the linked list of input events created with **InvertString()**.
AddIEvents()	Converts a list of input events to **CxMessages** and puts them into the network.

Chapter 32
EXPANSION LIBRARY

Amiga RAM expansion boards and other expansion bus peripherals are designed to reside at dynamically assigned address spaces within the system. The configuration and initialization of these expansion peripherals is performed by the *expansion.library*.

AUTOCONFIG™

The Amiga AUTOCONFIG protocol is designed to allow the dynamic assignment of available address space to expansion boards, eliminating the need for user configuration via jumpers. Such expansion boards include memory boards, hard disk controllers, network interfaces, and other special purpose expansion devices. Some expansion devices, such as RAM boards, require no special driver software. Other types of expansion devices may use a disk-loaded driver from the DEVS: or SYS:Expansion drawer, or an on-board ROM driver (for example, a self-booting hard disk controller).

This chapter will concentrate on the software and driver side of Zorro expansion devices, using a Zorro II device as an example. Zorro III devices have additional identifying bits and memory size options which are described in the Zorro III hardware documentation. For more information on Zorro II and Zorro III expansion hardware, see the ''Zorro Expansion Bus'' appendix of the *Amiga Hardware Reference Manual, 3rd Edition* from Addison-Wesley. For additional information specific to Zorro II boards, see the Commodore publication *A500/A2000 Technical Reference Manual.*

AUTOCONFIG occurs whenever the Amiga is powered on or reset. During early system initialization, expansion.library identifies the expansion boards that are installed in the Amiga and dynamically assigns an appropriate address range for each board to reside at. During this AUTOCONFIG process, each expansion board first appears in turn at $E80000 (Zorro II) or $FF000000 (Zorro III), presenting readable identification information, generally in a PAL or a ROM, at the beginning of the board. The identification includes the size of the board, its address space preferences, type of board (memory or other), and a unique Hardware Manufacturer Number assigned by Commodore Applications and Technical Support (CATS), West Chester, Pennsylvania.

The unique Hardware Manufacturer number, in combination with a vendor-supplied product number, provides a way for boards to be identified and for disk-based drivers to be matched with expansion boards. All expansion boards for the Amiga must implement the AUTOCONFIG protocol.

> *Note:* A unique Hardware Manufacturer number assigned by CATS is not the same as a Developer number.

The Expansion Sequence

During system initialization, expansion.library configures each expansion peripheral in turn by examining its identification information and assigning it an appropriate address space. If the board is a RAM board, it can be added to the system memory list to make the RAM available for allocation by system tasks.

Descriptions of all configured boards are kept in a private **ExpansionBase** list of **ConfigDev** structures. A board's identification information is stored in the **ExpansionRom** sub-structure contained within the **ConfigDev** structure. Applications can examine individual or all **ConfigDev** structures with the expansion.library function **FindConfigDev()**.

The **ConfigDev** structure is defined in *<libraries/configvars.h>* and *<.i>*:

```
struct ConfigDev {
    struct Node         cd_Node;
    UBYTE               cd_Flags;       /* (read/write) */
    UBYTE               cd_Pad;         /* reserved */
    struct ExpansionRom cd_Rom;         /* copy of board's expansion ROM */
    APTR                cd_BoardAddr;   /* where in memory the board was placed */
    ULONG               cd_BoardSize;   /* size of board in bytes */
    UWORD               cd_SlotAddr;    /* which slot number (PRIVATE) */
    UWORD               cd_SlotSize;    /* number of slots (PRIVATE) */
    APTR                cd_Driver;      /* pointer to node of driver */
    struct ConfigDev    *cd_NextCD;     /* linked list of drivers to config */
    ULONG               cd_Unused[4];   /* for whatever the driver wants */
};

/* cd_Flags */
#define CDB_SHUTUP      0       /* this board has been shut up */
#define CDB_CONFIGME    1       /* this board needs a driver to claim it */

#define CDF_SHUTUP      0x01
#define CDF_CONFIGME    0x02
```

The **ExpansionRom** structure within **ConfigDev** contains the board identification information that is read from the board's PAL or ROM at expansion time. The actual onboard identification information of a Zorro II board appears in the high nibbles of the first $40 words at the start of the board. Except for the first nibble pair ($00/$02) which when combined form **er_Type**, the information is stored in inverted (''ones-complement'') format where binary 1's are represented as 0's and 0's are represented as 1's. The expansion.library reads the nibbles of expansion information from the board, un-inverts them (except for $00/$02 **er_Type** which is already un-inverted), and combines them to form the elements of the **ExpansionRom** structure.

The **ExpansionRom** structure is defined in *<libraries/configregs.h>* and *<.i>*:

```
struct ExpansionRom {            /* First 16 bytes of the expansion ROM */
    UBYTE   er_Type;         /* Board type, size and flags */
    UBYTE   er_Product;      /* Product number, assigned by manufacturer */
    UBYTE   er_Flags;        /* Flags */
    UBYTE   er_Reserved03;   /* Must be zero ($ff inverted) */
    UWORD   er_Manufacturer; /* Unique ID,ASSIGNED BY COMMODORE-AMIGA! */
    ULONG   er_SerialNumber; /* Available for use by manufacturer */
    UWORD   er_InitDiagVec;  /* Offset to optional "DiagArea" structure */
    UBYTE   er_Reserved0c;
    UBYTE   er_Reserved0d;
    UBYTE   er_Reserved0e;
    UBYTE   er_Reserved0f;
};
```

SIMPLE EXPANSION LIBRARY EXAMPLE

The following example uses **FindConfigDev()** to print out information about all of the configured expansion peripherals in the system. **FindConfigDev()** searches the system's list of **ConfigDev** structures and returns a pointer to the **ConfigDev** structure matching a specified board:

```
newconfigdev = struct ConfigDev *FindConfigDev( struct ConfigDev *oldconfigdev,
                                                 LONG manufacturer, LONG product )
```

The **oldconfigdev** argument may be set to NULL to begin searching at the top of the system list or, if it points to a valid **ConfigDev**, searching will begin after that entry in the system list. The **manufacturer** and **product** arguments can be set to search for a specific manufacturer and product by number, or, if these are set to -1, the function will match any board.

```
;/* findboards.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 findboards.c
Blink FROM LIB:c.o,findboards.o TO findboards LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/configvars.h>

#include <clib/exec_protos.h>
#include <clib/expansion_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

struct Library   *ExpansionBase = NULL;

void main(int argc, char **argv)
{
    struct ConfigDev *myCD;
    UWORD m,i;
    UBYTE p,f,t;

    if((ExpansionBase=OpenLibrary("expansion.library",0L))==NULL)
      exit(RETURN_FAIL);

    /*---------------------------------------------------*/
    /* FindConfigDev(oldConfigDev,manufacturer,product) */
    /* oldConfigDev = NULL for the top of the list      */
    /* manufacturer = -1 for any manufacturer           */
    /* product      = -1 for any product                */
    /*---------------------------------------------------*/
```

```
    myCD = NULL;
    while(myCD=FindConfigDev(myCD,-1L,-1L)) /* search for all ConfigDevs */
      {
      printf("\n---ConfigDev structure found at location $%lx---\n",myCD);

      /* These values were read directly from the board at expansion time */
      printf("Board ID (ExpansionRom) information:\n");

      t = myCD->cd_Rom.er_Type;
      m = myCD->cd_Rom.er_Manufacturer;
      p = myCD->cd_Rom.er_Product;
      f = myCD->cd_Rom.er_Flags;
      i = myCD->cd_Rom.er_InitDiagVec;

      printf("er_Manufacturer         =%d=$%04x=(~$%4x)\n",m,m,(UWORD)~m);
      printf("er_Product              =%d=$%02x=(~$%2x)\n",p,p,(UBYTE)~p);

      printf("er_Type                 =$%02x",myCD->cd_Rom.er_Type);
      if(myCD->cd_Rom.er_Type & ERTF_MEMLIST)
          printf("  (Adds memory to free list)\n");
      else printf("\n");

      printf("er_Flags                =$%02x=(~$%2x)\n",f,(UBYTE)~f);
      printf("er_InitDiagVec          =$%04x=(~$%4x)\n",i,(UWORD)~i);


      /* These values are generated when the AUTOCONFIG(tm) software
       * relocates the board
       */
      printf("Configuration (ConfigDev) information:\n");
      printf("cd_BoardAddr            =$%lx\n",myCD->cd_BoardAddr);
      printf("cd_BoardSize            =$%lx (%ldK)\n",
             myCD->cd_BoardSize,((ULONG)myCD->cd_BoardSize)/1024);

      printf("cd_Flags                =$%x",myCD->cd_Flags);
      if(myCD->cd_Flags & CDF_CONFIGME)
          printf("\n");
      else printf("  (driver clears CONFIGME bit)\n");
      }
    CloseLibrary(ExpansionBase);
}
```

Expansion Board Drivers

The Amiga operating system contains support for matching up disk-based drivers with AUTOCONFIG boards. Though such drivers are commonly Exec devices, this is not required. The driver may, for instance, be an Exec library or task. Since 1.3, the system software also supports the initialization of onboard ROM driver software.

DISK BASED DRIVERS

Disk-based expansion board drivers and their icons are generally placed in the SYS:Expansion drawer of the user's SYS: disk or partition. The icon Tool Type field must contain the unique Hardware Manufacturer number, and the Product number of the expansion board(s) the driver is written for. (For more about icon Tool Type fields refer to the chapter on ''Workbench and Icon Library''.)

The BindDrivers command issued during the disk startup-sequence attempts to match disk-based drivers with their expansion boards. To do this, BindDrivers looks in the Tool Types field of all icon files in SYS:Expansion. If the Tool Type "PRODUCT" is found in the icon, then this is an icon file for a driver. Binddrivers will then attempt to match the manufacturer and product number in this PRODUCT Tool Type with those of a board that was configured at expansion time.

For example, suppose you are manufacturer #1019. You have two products, #1 and #2 which both use the same driver. The icon for your driver for these two products would have a Tool Type set to "PRODUCT=1019/1|1019/2". This means: I am an icon for a driver that works with product number 1 or 2 from manufacturer 1019, now bind me. Spaces are not legal. Here are two other examples:

```
PRODUCT=1208/11      is the Tool Type for a driver for product
                     11 from manufacturer number 1208.

PRODUCT=1017         is the Tool Type for a driver for any
                     product from manufacturer number 1017.
```

If a matching board is found for the disk-based driver, the driver code is loaded and then initialized with the Exec **InitResident()** function. From within its initialization code, the driver can get information about the board it is bound to by calling the expansion.library function **GetCurrentBinding()**. This function will provide the driver with a copy of a **CurrentBinding** structure, including a pointer to a **ConfigDev** structure (possibly linked to additional **ConfigDevs** via the **cd_NextCD** field) of the expansion board(s) that matched the manufacturer and product IDs.

```
/* this structure is used by GetCurrentBinding() and SetCurrentBinding() */
struct CurrentBinding {
    struct ConfigDev *cb_ConfigDev;         /* first configdev in chain */
    UBYTE *          cb_FileName;           /* file name of driver */
    UBYTE *          cb_ProductString;      /* product # string */
    UBYTE **         cb_ToolTypes;          /* tooltypes from disk object */
};
```

GetCurrentBinding() allows the driver to find out the base address and other information about its board(s). The driver must unset the CONFIGME bit in the **cd_Flags** field of the **ConfigDev** structure for each board it intends to drive, and record the driver's Exec node pointer in the **cd_Driver** structure. This node should contain the LN_NAME and LN_TYPE (i.e., NT_DEVICE, NT_TASK, etc.) of the driver.

> *Important Note:* The **GetCurrentBinding()** function, and driver binding in general, must be bracketed by an **ObtainConfigBinding()** and **ReleaseConfigBinding()** semaphore. The **BindDrivers** command obtains this semaphore and performs a **SetCurrentBinding()** before calling **InitResident()**, allowing the driver to simply do a **GetCurrentBinding()**.

Full source code for a disk-based Expansion or DEVS: sample device driver may be found in the Addison-Wesley *Amiga ROM Kernel Reference Manual: Devices.* Autodocs for expansion.library functions may be found in the Addison-Wesley *Amiga ROM Kernel Reference Manual: Includes and Autodocs.*

EXPANSION DRIVERS AND DOS

Two other expansion.library functions commonly used by expansion board drivers are **MakeDosNode()** and **AddDosNode()**. These functions allow a driver to create and add a DOS device node (for example DH0:) to the system. A new function, **AddBootNode()**, is also available in Release 2 (V36 and later versions of the OS) that can be used to add an autobooting DOS device node.

MakeDosNode() requires an initialized structure of environment information for creating a DOS device node. The format of the function is:

```
struct DeviceNode *deviceNode = MakeDosNode(parameterPkt);
```

The **parameterPkt** argument is a pointer (passed in A0 from assembler) to an initialized packet of environment parameters.

The parameter packet for **MakeDosNode()** consists of four longwords followed by a **DosEnvec** structure:

```
;---------------------------------------------------------------------
;
; Layout of parameter packet for MakeDosNode
;
;---------------------------------------------------------------------

* The packet for MakeDosNode starts with the following four
* longwords, directly followed by a DosEnvec structure.

    APTR  dosName           ; Points to a DOS device name (ex. 'RAM1',0)
    APTR  execName          ; Points to device driver name (ex. 'ram.device',0)
    ULONG unit              ; Unit number
    ULONG flags             ; OpenDevice flags

* The DosEnvec disk "environment" is a longword array that describes the
* disk geometry.  It is variable sized, with the length at the beginning.
* Here are the constants for a standard geometry.
* See libraries/filehandler.i for additional notes.

 STRUCTURE DosEnvec,0
    ULONG de_TableSize      ; Size of Environment vector
    ULONG de_SizeBlock      ; in longwords: standard value is 128
    ULONG de_SecOrg         ; not used; must be 0
    ULONG de_Surfaces       ; # of heads (surfaces). drive specific
    ULONG de_SectorPerBlock ; not used; must be 1
    ULONG de_BlocksPerTrack ; blocks per track. drive specific
    ULONG de_Reserved       ; DOS reserved blocks at start of partition.
    ULONG de_PreAlloc       ; DOS reserved blocks at end of partition
    ULONG de_Interleave     ; usually 0
    ULONG de_LowCyl         ; starting cylinder. typically 0
    ULONG de_HighCyl        ; max cylinder. drive specific
    ULONG de_NumBuffers     ; Initial # DOS of buffers.
    ULONG de_BufMemType     ; type of mem to allocate for buffers
    ULONG de_MaxTransfer    ; Max number of bytes to transfer at a time
    ULONG de_Mask           ; Address Mask to block out certain memory
    LONG  de_BootPri        ; Boot priority for autoboot
    ULONG de_DosType        ; ASCII (HEX) string showing filesystem type;
                            ; 0X444F5300 is old filesystem,
                            ; 0X444F5301 is fast file system
    ULONG de_Baud           ; Baud rate for serial handler
    ULONG de_Control        ; Control word for handler/filesystem
                            ; (used as filesystem/handler desires)
    ULONG de_BootBlocks     ; Number of blocks containing boot code
                            ; (for non-AmigaDOS filesystems)
    LABEL DosEnvec_SIZEOF
```

After making a DOS device node, drivers (except for autoboot drivers) use **AddDosNode(deviceNode)** to add their node to the system. Autoboot drivers will instead use the new Release 2 expansion.library **AddBootNode()** function (if running under V36 or higher) or the Exec **Enqueue()** function (if running under pre-V36) to add a **BootNode** to the **ExpansionBase.eb_MountList**.

ROM BASED AND AUTOBOOT DRIVERS

Since 1.3, the system software supports the initialization of ROM drivers residing on expansion peripherals, including the ability for drivers to provide a DOS node which the system can boot from. This feature is known as *Autoboot*.

Automatic boot from a ROM-equipped expansion board is accomplished before DOS is initialized. This facility makes it possible to automatically boot from a hard disk without any floppy disks inserted. Likewise, it is possible to automatically boot from any device which supports the ROM protocol, even allowing the initialization of a disk operating system other than the Amiga's dos.library. ROM-based drivers contain several special entry points that are called at different stages of system initialization. These three stages are known as DIAG, ROMTAG INIT and BOOT.

Events At DIAG Time

When your AUTOCONFIG hardware board is configured by the expansion initialization routine, its **ExpansionRom** structure is copied into the **ExpansionRom** subfield of a **ConfigDev** structure. This **ConfigDev** structure will be linked to the expansion.library's private list of configured boards.

After the board is configured, the **er_Type** field of its **ExpansionRom** structure is checked. The DIAGVALID bit set declares that there is a valid **DiagArea** (a ROM/diagnostic area) on this board. If there is a valid **DiagArea**, expansion next tests the **er_InitDiagVec** vector in its copy of the **ExpansionRom** structure. This offset is added to the base address of the configured board; the resulting address points to the start of this board's **DiagArea**.

```
struct ExpansionRom
{
    UBYTE   er_Type;            /* <-- if ERTB_DIAGVALID set */
    UBYTE   er_Product;
    UBYTE   er_Flags;
    UBYTE   er_Reserved03;
    UWORD   er_Manufacturer;
    ULONG   er_SerialNumber;
    UWORD   er_InitDiagVec;     /* <-- then er_InitDiagVec      */
    UBYTE   er_Reserved0c;      /*     is added to cd_BoardAddr */
    UBYTE   er_Reserved0d;      /*     and points to DiagArea   */
    UBYTE   er_Reserved0e;
    UBYTE   er_Reserved0f;
};
```

Now expansion knows that there is a **DiagArea**, and knows where it is.

```
struct DiagArea
{
    UBYTE   da_Config;          /* <-- if DAC_CONFIGTIME is set */
    UBYTE   da_Flags;
    UWORD   da_Size;            /* <-- then da_Size bytes will  */
    UWORD   da_DiagPoint;       /*     be copied into RAM       */
    UWORD   da_BootPoint;
    UWORD   da_Name;
    UWORD   da_Reserved01;
    UWORD   da_Reserved02;
};

/* da_Config definitions */
#define DAC_BUSWIDTH    0xC0    /* two bits for bus width */
#define DAC_NIBBLEWIDE  0x00
#define DAC_BYTEWIDE    0x40    /* invalid for 1.3 - see note below */
#define DAC_WORDWIDE    0x80

#define DAC_BOOTTIME    0x30    /* two bits for when to boot */
#define DAC_NEVER       0x00    /* obvious */
#define DAC_CONFIGTIME  0x10    /* call da_BootPoint when first
                                   configging the device */
#define DAC_BINDTIME    0x20    /* run when binding drivers to boards */
```

Next, expansion tests the first byte of the **DiagArea** structure to determine if the CONFIGTIME bit is set. If this bit is set, it checks the **da_BootPoint** offset vector to make sure that a valid bootstrap routine exists. If so, expansion copies **da_Size** bytes into RAM memory, starting at beginning of the **DiagArea** structure.

The copy will include the **DiagArea** structure itself, and typically will also include the **da_DiagPoint** ROM/diagnostic routine, a **Resident** structure (romtag), a device driver (or at least the device initialization tables or structures which need patching), and the **da_BootPoint** routine. In addition, the **BootNode** and parameter packet for **MakeDosNode()** may be included in the copy area for Diag-time patching. Strings such as DOS and Exec device names, library names, and the romtag ID string may also be included in the copy area so that both position-independent ROM code and position-independent routines in the copy area may reference them PC relative.

The copy will be made either nibblewise, or wordwise, according to the BUSWIDTH subfield of **da_Config**. Note that the **da_BootPoint** offset must be non-NULL, or else no copy will occur. (Note - under 1.3, DAC_BYTEWIDE is not supported. Byte wide ROMs must use DAC_NIBBLEWIDE and drivers must supply additional code to re-copy their **DiagArea**)

The following illustrates an example Diag copy area, and specifies the various fields which should be coded as relative offsets for later patching by your **DiagPoint** routine.

```
                    Example DiagArea Copy in RAM

DiagStart:          ; a struct DiagArea
            CCFF    ; da_Config, da_Flags
            SIZE    ; da_Size          - coded as EndCopy-DiagStart
            DIAG    ; da_DiagPoint     - coded as DiagEntry-DiagStart
            BOOT    ; da_BootPoint     - coded as BootEntry-DiagStart
            NAME    ; da_Name          - coded as DevName-DiagStart
            0000    ; da_Reserved01    - Above fields above are supposed
            0000    ; da_Reserved02      to be relative. No patching needed

Romtag:     rrrr    ; a struct Resident (''Romtag'')
            ...       RT_MATCHTAG, RT_ENDSKIP, RT_NAME and  RT_IDSTRING
            ...       addresses are coded relatively as label-DiagStart.
            ...       The RT_INIT vector is coded as a relative offset
            ...       from the start of the ROM.  DiagEntry patches these.

DevName:    ssss..0 ; The name string for the exec device
IdString:   iiii..0 ; The ID string for the Romtag

BootEntry:  BBBB    ; Boot-time code
            ...

DiagEntry:  DDDD    ; Diag-time code (position independent)
            ...       When called, performs patching of the relative-
                      coded addresses which need to be absolute.
OtherData:
            dddd    ; Device node or structs/tables (patch names, vectors)
            bbbb    ; BootNode (patch ln_Name and bn_DeviceNode)
            pppp    ; MakeDosNode packet (patch dos and exec names)

            ssss    ; other name, ID, and library name strings
            ...
EndCopy:
```

Now the ROM "image" exists in RAM memory. Expansion stores the ULONG address of that "image" in the UBYTES **er_Reserved0c, 0d, 0e** and **0f**. The address is stored with the most significant byte in **er_Reserved0c**, the next to most significant byte in **er_Reserved0d**, the next to least significant byte in **er_Reserved0e**, and the least significant byte in **er_Reserved0f** - i.e., it is stored as a longword at the address **er_Reserved0c**.

Expansion finally checks the **da_DiagPoint** offset vector, and if valid executes the ROM/diagnostic routine contained as part of the ROM "image". This diagnostic routine is responsible for patching the ROM image so that required absolute addresses are relocated to reflect the actual location of code and strings, as well as performing any diagnostic functions essential to the operation of its associated AUTOCONFIG board. The structures which require patching are located within the copy area so that they can be patched at this time. Patching is required because many of the structures involved require absolute pointers to such things as name strings and code, but the absolute locations of the board and the RAM copy area are not known when code the structures.

The patching may be accomplished by coding pointers which require absolute addresses instead as relative offsets from either the start of the **DiagArea** structure, or the start of the board's ROM (depending on whether the final absolute pointer will point to a RAM or ROM location). The Diag routine is passed both the actual base address of the board, and the address of the Diag copy area in RAM. The routine can then patch the structures in the Diag copy area by adding the appropriate address to resolve each pointer.

Example **DiagArea** and Diag patching routine:

```
**
** Sample autoboot code fragment
**
** These are the calling conventions for the Diag routine
**
** A7 -- points to at least 2K of stack
** A6 -- ExecBase
** A5 -- ExpansionBase
** A3 -- your board's ConfigDev structure
** A2 -- Base of diag/init area that was copied
** A0 -- Base of your board
**
** Your Diag routine should return a non-zero value in D0 for success.
** If this value is NULL, then the diag/init area that was copied
** will be returned to the free memory pool.
**

            INCLUDE "exec/types.i"
            INCLUDE "exec/nodes.i"
            INCLUDE "exec/resident.i"
            INCLUDE "libraries/configvars.i"

            ; LVO's resolved by linking with library amiga.lib
            XREF    _LVOFindResident

ROMINFO     EQU      1
ROMOFFS     EQU     $0

* ROMINFO defines whether you want the AUTOCONFIG information in
* the beginning of your ROM (set to 0 if you instead have PALS
* providing the AUTOCONFIG information instead)
*
* ROMOFFS is the offset from your board base where your ROMs appear.
* Your ROMs might appear at offset 0 and contain your AUTOCONFIG
* information in the high nibbles of the first $40 words ($80 bytes).
* Or, your autoconfig ID information may be in a PAL, with your
* ROMs possibly being addressed at some offset (for example $2000)
* from your board base.  This ROMOFFS constant will be used as an
* additional offset from your configured board address when patching
* structures which require absolute pointers to ROM code or data.

*----- We'll store Version and Revision in serial number
VERSION     EQU   37          ; also the high word of serial number
REVISION    EQU   1           ; also the low word of serial number

* See the Addison-Wesley Amiga Hardware Manual for more info.

MANUF_ID    EQU   2011        ; CBM assigned (2011 for hackers only)
PRODUCT_ID  EQU   1           ; Manufacturer picks product ID

BOARDSIZE   EQU   $40000          ; How much address space board decodes
SIZE_FLAG   EQU   3           ; Autoconfig 3-bit flag for BOARDSIZE
                              ;    0=$800000(8meg)  4=$80000(512K)
                              ;    1=$10000(64K)    5=$100000(1meg)
                              ;    2=$20000(128K)   6=$200000(2meg)
                              ;    3=$40000(256K)   7=$400000(4meg)
            CODE

*******  RomStart  ***************************************************
**********************************************************************

RomStart:

      IFGT    ROMINFO
```

```
;
;   ExpansionRom structure
;
;   Note - If you implement your ExpansionRom and ExpansionControl
;   with PALS, then you can comment out everything until DiagStart:
;   (ie. Make ROMID EQU 0)

*    ; High nibbles of first two words ($00,$02) are er_Type (not inverted)
                                ; er_Type
        dc.w  $D000             ;   11xx normal board type
                                ;   xx0x not in memory free list
                                ;   xxx1 Diag valid (has driver)
        dc.w  (SIZE_FLAG<<12)&$7000  ;   0xxx not chained
                                ;   xnnn flags board size

*    ; High nibbles of next two words are er_Product
*    ; These are inverted (~), as are all other words except $40 and $42

                                ; er_Product
        dc.w  (~(PRODUCT_ID<<8))&$f000,(~(PRODUCT_ID<<12))&$f000

                                ; er_Flags
        dc.w  (~$C000)&$f000        ;   ~1xxx board is moveable
                                ;   ~x1xx board can't be shut up
        dc.w  (~0)&$f000        ;

        dc.w  (~0)&$f000,(~0)&$f000   ; er_Reserved03

                                ; er_Manufacturer
        dc.w  (~(MANUF_ID))&$f000,(~(MANUF_ID<<4))&$f000
        dc.w  (~(MANUF_ID<<8))&$f000,(~(MANUF_ID<<12))&$f000

                                ; er_SerialNumber
        dc.w  (~(VERSION))&$f000,(~(VERSION<<4))&$f000
        dc.w  (~(VERSION<<8))&$f000,(~(VERSION<<12))&$f000
        dc.w  (~(REVISION))&$f000,(~(REVISION<<4))&$f000
        dc.w  (~(REVISION<<8))&$f000,(~(REVISION<<12))&$f000

                                ; er_InitDiagVec
        dc.w  (~((DiagStart-RomStart)))&$f000
        dc.w  (~((DiagStart-RomStart)<<4))&$f000
        dc.w  (~((DiagStart-RomStart)<<8))&$f000
        dc.w  (~((DiagStart-RomStart)<<12))&$f000

        dc.w  (~0)&$f000,(~0)&$f000   ; er_Reserved0c
        dc.w  (~0)&$f000,(~0)&$f000   ; er_Reserved0d
        dc.w  (~0)&$f000,(~0)&$f000   ; er_Reserved0e
        dc.w  (~0)&$f000,(~0)&$f000   ; er_Reserved0f

    IFNE  *-RomStart-$40
    FAIL  "ExpansionRom structure not the right size"
    ENDC

        ;Note: nibbles $40 and $42 are not to be inverted
        dc.w  (0)&$f000,(0)&$f000     ; ec_Interrupt (no interrupts)
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved11
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_BaseAddress (write only)
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Shutup (write only)
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved14
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved15
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved16
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved17
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved18
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved19
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1a
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1b
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1c
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1d
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1e
        dc.w  (~0)&$f000,(~0)&$f000   ; ec_Reserved1f

    IFNE  *-RomStart-$80
    FAIL  "Expansion Control structure not the right size"
    ENDC

    ENDC  ;ROMINFO
```

```
*******  DiagStart  ************************************************
DiagStart:  ; This is the DiagArea structure whose relative offset from
            ; your board base appears as the Init Diag vector in your
            ; autoconfig ID information.  This structure is designed
            ; to use all relative pointers (no patching needed).
            dc.b    DAC_WORDWIDE+DAC_CONFIGTIME    ; da_Config
            dc.b    0                              ; da_Flags
            dc.w    EndCopy-DiagStart              ; da_Size
            dc.w    DiagEntry-DiagStart            ; da_DiagPoint
            dc.w    BootEntry-DiagStart            ; da_BootPoint
            dc.w    DevName-DiagStart              ; da_Name
            dc.w    0                              ; da_Reserved01
            dc.w    0                              ; da_Reserved02

*******  Resident Structure  ****************************************
Romtag:
            dc.w    RTC_MATCHWORD      ; UWORD RT_MATCHWORD
rt_Match:   dc.l    Romtag-DiagStart   ; APTR  RT_MATCHTAG
rt_End:     dc.l    EndCopy-DiagStart  ; APTR  RT_ENDSKIP
            dc.b    RTW_COLDSTART      ; UBYTE RT_FLAGS
            dc.b    VERSION            ; UBYTE RT_VERSION
            dc.b    NT_DEVICE          ; UBYTE RT_TYPE
            dc.b    20                 ; BYTE  RT_PRI
rt_Name:    dc.l    DevName-DiagStart  ; APTR  RT_NAME
rt_Id:      dc.l    IdString-DiagStart ; APTR  RT_IDSTRING
rt_Init:    dc.l    Init-RomStart      ; APTR  RT_INIT


******* Strings referenced in Diag Copy area  ***********************
DevName:    dc.b    'abc.device',0                  ; Name string
IdString    dc.b    'abc ',48+VERSION,'.',48+REVISION   ; Id string

DosName:    dc.b    'dos.library',0               ; DOS library name

DosDevName: dc.b    'ABC',0        ; dos device name for MakeDosNode()
                                   ;   (dos device will be ABC:)

            ds.w    0              ; word align

*******  DiagEntry  *************************************************
********************************************************************
*
*   success = DiagEntry(BoardBase,DiagCopy, configDev)
*   d0                  a0        a2                 a3
*
*   Called by expansion architecture to relocate any pointers
*   in the copied diagnostic area.   We will patch the romtag.
*   If you have pre-coded your MakeDosNode packet, BootNode,
*   or device initialization structures, they would also need
*   to be within this copy area, and patched by this routine.
*
********************************************************************

DiagEntry:
            lea     patchTable-RomStart(a0),a1   ; find patch table
            adda.l  #ROMOFFS,a1                  ; adjusting for ROMOFFS

* Patch relative pointers to labels within DiagCopy area
* by adding Diag RAM copy address.  These pointers were coded as
* long relative offsets from base of the DiagArea structure.
*
dpatches:
            move.l  a2,d1           ;d1=base of ram Diag copy
dloop:
            move.w  (a1)+,d0        ;d0=word offs. into Diag needing patch
            bmi.s   bpatches        ;-1 is end of word patch offset table
            add.l   d1,0(a2,d0.w)   ;add DiagCopy addr to coded rel. offset
            bra.s   dloop

* Patches relative pointers to labels within the ROM by adding
* the board base address + ROMOFFS.  These pointers were coded as
* long relative offsets from RomStart.
*
bpatches:
            move.l  a0,d1           ;d1 = board base address
            add.l   #ROMOFFS,d1     ;add offset to where your ROMs are
```

```
rloop:
           move.w   (a1)+,d0          ;d0=word offs. into Diag needing patch
           bmi.s    endpatches        ;-1 is end of patch offset table
           add.l    d1,0(a2,d0.w)     ;add ROM address to coded relative offset
           bra.s    rloop

endpatches:
           moveq.l #1,d0             ; indicate "success"
           rts

*******  BootEntry  **************************************************
**********************************************************************

BootEntry: lea      DosName(PC),a1          ; 'dos.library',0
           jsr      _LVOFindResident(a6)    ; find the DOS resident tag
           move.l   d0,a0                   ; in order to bootstrap
           move.l   RT_INIT(A0),a0          ; set vector to DOS INIT
           jsr      (a0)                    ; and initialize DOS
           rts

*
* End of the Diag copy area which is copied to RAM
*
EndCopy:
*************************************************************************

*************************************************************************
*
*   Beginning of ROM driver code and data that is accessed only in
*   the ROM space.  This must all be position-independent.
*

patchTable:
* Word offsets into Diag area where pointers need Diag copy address added
           dc.w   rt_Match-DiagStart
           dc.w   rt_End-DiagStart
           dc.w   rt_Name-DiagStart
           dc.w   rt_Id-DiagStart
           dc.w   -1

* Word offsets into Diag area where pointers need boardbase+ROMOFFS added
           dc.w   rt_Init-DiagStart
           dc.w   -1

*******  Romtag InitEntry  **********************************************
*************************************************************************

Init:      ; After Diag patching, our romtag will point to this
           ; routine in ROM so that it can be called at Resident
           ; initialization time.
           ; This routine will be similar to a normal expansion device
           ; initialization routine, but will MakeDosNode then set up a
           ; BootNode, and Enqueue() on eb_MountList.
           ;
           rts

           ; Rest of your position-independent device code goes here.

           END
```

Your **da_DiagPoint** ROM/diagnostic routine should return a non-zero value to indicate success; otherwise the ROM "image" will be unloaded from memory, and its address will be replaced with NULL bytes in locations **er_Reserved0c, 0d, 0e** and **0f**.

Now that the ROM "image" has been copied into RAM, validated, and linked to board's **ConfigDev** structure, the expansion module is free to configure all other boards on the utility.library's private list of boards.

It may help to see just how a card's ROM AUTOCONFIG information corresponds to the **ExpansionRom** structure. This chart shows the contents of on-card memory for a fictional expansion card. Note that the ExpansionRom.Flags field ($3F in addresses $08/$0A below) is shown interpreted in its inverted form of $3F. Once the value is uninverted to become $C0, you should use the #defines in *<libraries/configregs.h>* to interpret it.

Table 32-1: Sample Zorro II AUTOCONFIG ROM Information Viewed as a Hex Dump

```
            FLAG AND FIELD DEFINITIONS                THIS BOARD
-----------------------------------------------------------------

          1xxx chained                           11 = Normal type
          x111 size                              Don't addmem
          000=8meg,001=64K,010=128K,etc.         ROM Vector Valid
 11xx type      /          1xxx nopref           Not chained
 xx1x addmem   /           x1xx canshut          Size 256K
 xxx1 ROM     /            xx11 reserved         Product#=~$FE=1
    \        / ~Prod#       /                    Flags=~$3F=$C0
     \      /   /  \       /  res. reserved      Prefer exp space
0000: D0003000 F000E000 3000F000 F000F000        Can't be shut up
-----------------------------------------------------------------
      ~Manufacturer#     ~HiWord Serial#     Manu#=~$F824=$7DB=2011
       /   /  \   \       /   /   \   \      HiSer=~$FFDA=$0025=37
0010: F0008000 20004000 F000F000 D000A000
-----------------------------------------------------------------
      ~LoWord Serial#     ~Rom Vector        LoSer=~$FFFE=$0001=1
       /   /  \ \       /   /   \   \      Rom Vector=~$FF7F=$80
0020: F000F000 F000E000 F000F000 7000F000         from board base
-----------------------------------------------------------------
```

The AUTOCONFIG information from the above card would appear as follows in an **ExpansionRom** structure:

Nibble Pairs	ExpansionRom Field	Value
00/02	er_Type	$D3
04/06	er_Product	$01 = 1
08/0A	er_Flags	$C0
10/12 and 14/16	er_Manufacturer	$07DB = 2011
18/1A thru 24/26	er_SerialNumber	$00250001
28/2A and 2C/2E	er_InitDiagVec	$0080

If a card contains a ROM driver (Rom Vector valid), and the vector is at offset $80 (as in this example) the **DiagArea** structure will appear at offset $0080 from the base address of the board. This example card's **Resident** structure (romtag) directly follows its **DiagArea** structure.

```
   WORDWIDE+CONFIGTIME             ROMTAG
    \ flags     DiagPt   Devname     starts
     \ \ DAsize  / BootPt /          here        DiagPt, BootPt,
      \ \  /\    /\   /\    /\ res. res. /\      DevName relative
0080: 90000088 004A0076 00280000 00004AFC        to Diag struct
-----------------------------------------------------------------
                  COLDSTART  NT_DEVICE           backptr,endskip,
                      \ ver  /priority           and DevName coded
      backptr  endskip  \ \  / /  DevName        relative, patched
0090: 0000000E 00000088 01250314 00000028        at Diag time
-----------------------------------------------------------------
                                                 ID and InitEntry
      IDstring InitEntry                         coded relative,
00A0: 00000033 00000116                          patched at Diag
-----------------------------------------------------------------
```

Events At ROMTAG INIT Time

Next, most resident system modules (for example graphics) are initialized. As part of the system initialization procedure a search is made of the expansion.library's private list of boards (which contains a **ConfigDev** structure for each of the AUTOCONFIG hardware boards). If the **cd_Flags** specify CONFIGME and the **er_Type** specifies DIAGVALID, the system initialization will do three things:

First, it will set the current **ConfigDev** as the *current binding* (see the expansion.library **SetCurrentBinding()** function). Second, it will check the **DiagArea**'s **da_Config** flag to make sure that the CONFIGTIME bit is set. Third, it will search the ROM "image" associated with this hardware board for a valid **Resident** structure (*<exec/resident.h>*); and, if one is located, will call **InitResident()** on it, passing a NULL segment list pointer as part of the call.

Next, the board's device driver is initialized. The **Resident** structure associated with this board's device driver (which has now been patched by the ROM/diagnostic routine) should follow standard system conventions in initializing the device driver provided in the boot ROMs. This driver should obtain the address of its associated **ConfigDev** structure via **GetCurrentBinding()**.

Once the driver is initialized, it is responsible for some further steps. It must clear the CONFIGME bit in the **cd_Flags** of its **ConfigDev** structure, so that the system knows not to configure this device again if binddrivers is run after bootstrap. Also, though it is not currently mandatory, the driver should place a pointer to its Exec node in the **cd_Driver** field of the **ConfigDev** structure. This will generally be a device (NT_DEVICE) node. And for this device to be bootable, the driver must create a **BootNode** structure, and link this **BootNode** onto the expansion.library's **eb_MountList**.

The **BootNode** structure (see *<libraries/expansionbase.h>*) contains a **Node** of the new type NT_BOOTNODE (see *<exec/nodes.h>*). The driver *must* initialize the **ln_Name** field to point to the **ConfigDev** structure which it has obtained via the **GetCurrentBinding()** call. The **bn_Flags** subfield is currently unused and should be initialized to NULL. The **bn_DeviceNode** must be initialized to point to the **DosNode** for the device.

When the DOS is initialized later, it will attempt to boot from the first **BootNode** on the **eb_MountList**. The **eb_MountList** is a priority sorted **List**, with nodes of the highest priority at the head of the **List**. For this reason, the device driver must enqueue a **BootNode** onto the list using the Exec library function **Enqueue()**.

In the case of an autoboot of AmigaDOS, the **BootNode** must be linked to a **DeviceNode** of the AmigaDOS type (see *<libraries/filehandler.h>*), which the driver can create via the expansion library **MakeDosNode()** function call. When the DOS "wakes up", it will attempt to boot from this **DeviceNode**.

Events At BOOT Time

If there is no boot disk in the internal floppy drive, the system strap module will call a routine to perform autoboot. It will examine the **eb_MountList**; find the highest priority **BootNode** structure at the head of the List; validate the **BootNode**; determine which **ConfigDev** is associated with this **BootNode**; find its **DiagArea**; and call its **da_BootPoint** function in the ROM "image" to bootstrap the appropriate DOS. Generally, the **BootPoint** code of a ROM driver will perform the same function as the boot code installed on a floppy disk, i.e., it will **FindResident()** the dos.library, and jump to its RT_INIT vector. The **da_BootPoint** call, if successful, should not return.

If a boot disk *is* in the internal floppy drive, the system strap will **Enqueue()** a **BootNode** on the **eb_MountList** for DF0: at the suggested priority (see the Autodoc for the expansion.library **AddDosNode()** function). Strap will then open AmigaDOS, overriding the autoboot. AmigaDOS will boot from the highest priority node on the **eb_MountList** which should, in this case, be DF0:. Thus, games and other bootable floppy disks will still be able to obtain the system for their own use.

In the event that there is no boot disk in the internal floppy drive *and* there are no ROM bootable devices on the autoconfiguration chain, the system does the normal thing, asking the user to insert a Workbench disk, and waiting until its request is satisfied before proceeding.

RIGIDDISKBLOCK AND ALTERNATE FILESYSTEMS

Through the use of **RigidDiskBlock** information and the FileSys.resource, it is possible for an autoboot driver to have access to enough information to mount all of its device partitions and even load alternate filesystems for use with these partitions.

The **RigidDiskBlock** specification (also known as "hardblocks") defines blocks of data that exist on a hard disk to describe that disk. These blocks are created or modified with an installation utility (such as the hard drive *Prep* utility for the A2090A ST506/SCSI controller card) provided by the disk controller manufacturer, and they are read and used by the device driver ROM (or expansion) code. They are not generally accessible to the user as they do not appear on any DOS device. The blocks are tagged with a unique identifier, checksummed, and linked together.

The five block types currently defined are **RigidDiskBlock**, **BadBlockBlock**, **PartitionBlock**, **FileSysHeaderBlock**, and **LoadSegBlock**.

The root of these blocks is the **RigidDiskBlock**. The **RigidDiskBlock** must exist on the disk within the first RDB_LOCATION_LIMIT (16) blocks. This inhibits the use of the first cylinder(s) in an AmigaDOS partition: although it is strictly possible to store the **RigidDiskBlock** data in the reserved area of a partition, this practice is discouraged since the reserved blocks of a partition are overwritten by Format, Install, DiskCopy, etc. The recommended disk layout, then, is to use the first cylinder(s) to store all the drive data specified by these blocks: i.e. partition descriptions, file system load images, drive bad block maps, spare blocks, etc. This allocation range is described in the **RigidDiskBlock**.

The **RigidDiskBlock** contains basic information about the configuration of the drive: number and size of blocks, tracks, and cylinders, as well as other relevant information. The **RigidDiskBlock** points to bad block, partition, file system and drive initialization description blocks.

The **BadBlockBlock** list contains a series of bad-block/good-block pairs. Each block contains as many as will fit in a physical sector on the drive. These mappings are to be handled by the driver on read and write requests.

The drive initialization description blocks are **LoadSegBlocks** that are loaded at boot time to perform drive-specific initialization. They are called with both C-style parameters on the stack, and assembler parameters in registers as follows:

```
d0 = DriveInit(lun, rdb, ior)(d0/a0/a1)
```

where **lun** is the SCSI logical unit number (needed to construct SCSI commands), **rdb** is a pointer to a memory copy of the **RigidDiskBlock** (which should not be altered), and **ior** is a standard IO request block that can be used to access the drive with synchronous **DoIO()** calls.

The result of **DriveInit()** is either -1, 0, or 1. A -1 signifies that an error occurred and drive initialization cannot continue. A 0 (zero) result reports success. In cases -1 and 0, the code is unloaded. A result of 1 reports success, and causes the code to be kept loaded. Furthermore, this resident code will be called whenever a reset is detected on the SCSI bus.

The **FileSysHeaderBlock** entries contain code for alternate file handlers to be used by partitions. There are several strategies that can be used to determine which of them to load. The most robust would scan all drives for those that are both required by partitions and have the highest **fhb_Version**, and load those. Whatever method is used, the loaded file handlers are added to the Exec resource FileSystem.resource, where they are used as needed to mount disk partitions.

The **PartitionBlock** entries contains most of the data necessary to add each partition to the system. They replace the Mount and DEVS:MountList mechanism for adding these partitions. The only items required by the expansion.library **MakeDosNode()** function which are not in this partition block are the Exec device name and unit, which is expected to be known by driver reading this information. The file system to be used is specified in the **pb_Environment**. If it is not the default file system (i.e., as specified in a FileSystem.resource's **FileSysEntry** before adding it to the DOS list.

Only 512 byte blocks were supported by the pre-V36 file system, but this proposal was forward-looking by making the block size explicit, and by using only the first 256 bytes for all blocks but the **LoadSeg** and **BadBlock** data. Under the present filesystem, this allows using drives formatted with sectors 256 bytes or larger (i.e., 256, 512, 1024, etc). **LoadSeg** and **BadBlock** data use whatever space is available in a sector.

RigidDiskBlock

This is the current specification for the **RigidDiskBlock**:

```
rdb_ID              == 'RDSK'
rdb_SummedLongs     == 64

rdb_ChkSum          block checksum (longword sum to zero)

rdb_HostID          SCSI Target ID of host
                    This is the initiator ID of the creator of this
                    RigidDiskBlock.  It is intended that
                    modification of the RigidDiskBlock, or of any
                    of the blocks pointed to by it, by another
                    initiator (other than the one specified here)
                    be allowed only after a suitable warning.  The
                    user is then expected to perform an audio
                    lock out ("Hey, is anyone else setting up SCSI
                    stuff on this bus?").  The rdb_HostID may
                    become something other than the initiator ID
                    when connected to a real network: that is an
                    area for future standardization.

rdb_BlockBytes      size of disk blocks
                    Under pre-V36 filesystem, this must be 512 for
                    a disk with any AmigaDOS partitions on it.
                    Present filesystem supports 256, 512, 1024, etc.

rdb_Flags           longword of flags:

    RDBF._LAST          no disks exist to be configured after this
                        one on this controller (SCSI bus).

    RDBF._LASTLUN       no LUNs exist to be configured greater
                        than this one at this SCSI Target ID

    RDBF._LASTTID       no Target IDs exist to be configured
                        greater than this one on this SCSI bus
```

```
RDBF._NORESELECT    don't bother trying to perform reselection
                    when talking to this drive

RDBF._DISKID        rdb_Disk... identification variables below
                    contain valid data.

RDBF._CTRLRID       rdb_Controller... identification variables
                    below contain valid data.

RDBF._SYNCH         drive supports scsi synchronous mode
                    CAN BE DANGEROUS TO USE IF IT DOESN'T!
```

These fields point to other blocks on the disk which are not a part of any filesystem. All block pointers referred to are block numbers on the drive.

```
rdb_BadBlockList        optional bad block list
                        A singly linked list of blocks of type
                        PartitionBlock

rdb_PartitionList       optional first partition block
                        A singly linked list of blocks of type
                        PartitionBlock

rdb_FileSysHeaderList   optional file system header block
                        A singly linked list of blocks of type
                        FileSysHeaderBlock

rdb_DriveInit           optional drive-specific init code
                        A singly linked list of blocks of type
                        LoadSegBlock containing initialization code.
                        Called as DriveInit(lun,rdb,ior)(d0/a0/a1).

rdb_Reserved1[6]        set to $ffffffffs
                        These are reserved for future block lists.
                        Since NULL for block lists is $ffffffff, these
                        reserved entries must be set to $ffffffff.
```

These fields describe the physical layout of the drive.

```
rdb_Cylinders           number of drive cylinders
rdb_Sectors             sectors per track
rdb_Heads               number of drive heads

rdb_Interleave          interleave
                        This drive interleave is independent from, and
                        unknown to, the DOS's understanding of
                        interleave as set in the partition's
                        environment vector.

rdb_Park                landing zone cylinder
rdb_Reserved2[3]        set to zeros
```

These fields are intended for ST506 disks. They are generally unused for SCSI devices and set to zero.

```
rdb_WritePreComp        starting cylinder: write precompensation
rdb_ReducedWrite        starting cylinder: reduced write current
rdb_StepRate            drive step rate
rdb_Reserved3[5]        set to zeros
```

These fields are used while partitions are set up to constrain the partitionable area and help describe the relationship between the drive's logical and physical layout.

```
rdb_RDBlocksLo          low block of the range allocated for
                        blocks described here.  Replacement blocks
                        for bad blocks may also live in this range.
```

```
rdb_RDBlocksHi          high block of this range (inclusive)

rdb_LoCylinder          low cylinder of partitionable disk area
                        Blocks described by this include file will
                        generally be found in cylinders below this one.

rdb_HiCylinder          high cylinder of partitionable data area
                        Usually rdb_Cylinders-1.

rdb_CylBlocks           number of blocks available per cylinder
                        This may be rdb_Sectors*rdb_Heads, but a SCSI
                        disk that, for example, reserves one block per
                        cylinder for  bad block mapping would use
                        rdb_Sectors*rdb_Heads-1.

rdb_AutoParkSeconds     number of seconds to wait before parking
                        drive heads automatically.  If zero, this
                        feature is not desired.

rdb_HighRDSKBlock       highest block used by these drive definitions
                        Must be less than or equal to rdb_RDBBlocksHi.
                        All replacements for bad blocks should be
                        between rdb_HighRDSKBlock+1 and rdb_RDBBlocksHi
                        (inclusive).

rdb_Reserved4           set to zeros
```

These fields are of the form available from a SCSI Identify command. Their purpose is to help the user identify the disk during setup. Entries exist for both controller and disk for non-embedded SCSI disks.

```
rdb_DiskVendor          vendor name of the disk
rdb_DiskProduct         product name of the disk
rdb_DiskRevision        revision code of the disk
rdb_ControllerVendor    vendor name of the disk controller
rdb_ControllerProduct   product name of the disk controller
rdb_ControllerRevision  revision code of the disk controller
rdb_Reserved5[10]       set to zeros
```

BadBlockBlock

This is the current specification for the **BadBlockBlock**. The end of data occurs when **bbb_Next** is NULL ($FFFFFFFF), and the summed data is exhausted.

```
bbb_ID                  == 'BADB'

bbb_SummedLongs         size of this checksummed structure
                        Note that this is not 64 like most of the other
                        structures.  This is the number of valid longs
                        in this image, and can be from 6 to
                        rdb_BlockBytes/4.  The latter is the best size
                        for all blocks other than the last one.

bbb_ChkSum              block checksum (longword sum to zero)

bbb_HostID              SCSI Target ID of host
                        This describes the initiator ID for the creator
                        of these blocks.  (see rdb_HostID discussion)

bbb_Next                block number of the next BadBlockBlock
bbb_Reserved            set to zeros

bbb_BlockPairs[61]      pairs of block remapping information
                        The data starts here and continues as long as
                        indicated by bbb_SummedLongs-6: e.g. if
                        bbb_SummedLongs is 128 (512 bytes), 61 pairs
                        are described here.
```

PartitionBlock

This is the current specification for the **PartitionBlock**. Note that while reading these blocks you may encounter partitions that are not to be mounted because the **pb_HostID** does not match, or because the **pb_DriveName** is in use and no fallback strategy exists, or because PBF._NOMOUNT is set, or because the proper filesystem cannot be found. Some partitions may be mounted but not be bootable because PBF._BOOTABLE is not set.

```
pb_ID                 == 'PART'
pb_SummedLongs        == 64
pb_ChkSum             block checksum (longword sum to zero)

pb_HostID             SCSI Target ID of host
                      This describes the initiator ID for the owner
                      of this partition.  (see rdb_HostID discussion)

pb_Next               block number of the next PartitionBlock

pb_Flags              see below for defines

    PBF._BOOTABLE     this partition is intended to be bootable
                      (e.g. expected directories and files exist)

    PBF._NOMOUNT      this partition description is to reserve
                      space on the disk without mounting it.
                      It may be manually mounted later.
pb_Reserved1[2]       set to zeros

pb_DevFlags           preferred flags for OpenDevice
pb_DriveName          preferred DOS device name: BSTR form
                      This name is not to be used if it is already
                      in use.
```

Note that **pb_Reserved2** will always be at least 4 longwords so that the RAM image of this record may be converted to the parameter packet to the expansion.library function **MakeDosNode()**.

```
pb_Reserved2[15]      filler to make 32 longwords so far
```

The specification of the location of the partition is one of the components of the environment, below. If possible, describe the partition in a manner that tells the DOS about the physical layout of the partition: specifically, where the cylinder boundaries are. This allows the filesystem's smart block allocation strategy to work.

```
pb_Environment[17]        environment vector for this partition
                          containing:

    de_TableSize          size of Environment vector
    de_SizeBlock          == 128 (for 512 bytes/logical block)
    de_SecOrg             == 0
    de_Surfaces           number of heads (see layout discussion)
    de_SectorPerBlock     == 1
    de_BlocksPerTrack     blocks per track (see layout discussion)

    de_Reserved           DOS reserved blocks at start of partition.
                          Must be >= 1.  2 is recommended.

    de_PreAlloc           DOS reserved blocks at end of partition
                          Valid only for filesystem type DOS^A (the
                          fast file system).  Zero otherwise.

    de_Interleave         DOS interleave
                          Valid only for filesystem type DOS^@ (the
                          old file system).  Zero otherwise.

    de_LowCyl             starting cylinder
    de_HighCyl            max cylinder
    de_NumBuffers         initial # DOS of buffers.
```

```
    de_BufMemType          type of mem to allocate for buffers
                           The second argument to AllocMem().

    de_MaxTransfer         max number of bytes to transfer at a time.
                           Drivers should be written to handle requests
                           of any length.

    de_Mask                address mask to block out certain memory
                           Normally $00ffffff  for DMA devices.

    de_BootPri             Boot priority for autoboot
                           Suggested value: zero.  Keep less than
                           five, so it won't override a boot floppy.

    de_DosType             ASCII string showing filesystem type;
                           DOS^@ ($444F5300) is old filesystem,
                           DOS^A ($444F5301) is fast file system.
                           UNI<anything> is a Unix partition.

pb_EReserved[15]           reserved for future environment vector
```

FileSysHeaderBlock

The current specification for the **FileSysHeaderBlock** follows.

```
fhb_ID                     == 'FSHD'
fhb_SummedLongs            == 64
fhb_ChkSum                 block checksum (longword sum to zero)

fhb_HostID                 SCSI Target ID of host
                           This describes the initiator ID for the
                           creator of this block.  (see rdb_HostID
                           discussion)

fhb_Next                   block number of next FileSysHeaderBlock
fhb_Flags                  see below for defines
fhb_Reserved1[2]           set to zero
```

The following information is used to construct a **FileSysEntry** node in the FileSystem.resource.

```
fhb_DosType                file system description
                           This is matched with a partition environment's
                           de_DosType entry.

fhb_Version                release version of this load image
                           Usually MSW is version, LSW is revision.

fhb_PatchFlags             patch flags
                           These are bits set for those of the following
                           that need to be substituted into a standard
                           device node for this file system, lsb first:
                           e.g. 0x180 to substitute SegList & GlobalVec

fhb_Type                   device node type: zero
fhb_Task                   standard dos "task" field: zero
fhb_Lock                   not used for devices: zero
fhb_Handler                filename to loadseg: zero placeholder
fhb_StackSize              stacksize to use when starting task
fhb_Priority               task priority when starting task
fhb_Startup                startup msg: zero placeholder

fhb_SegListBlocks          first of linked list of LoadSegBlocks:
                           Note that if the fhb_PatchFlags bit for this
                           entry is set (bit 7), the blocks pointed to by
                           this entry must be LoadSeg'd and the resulting
                           BPTR put in the FileSysEntry node.

fhb_GlobalVec              BCPL global vector when starting task
                           Zero or -1.

fhb_Reserved2[23]          (those reserved by PatchFlags)
fhb_Reserved3[21]          set to zero
```

LoadSegBlock

This is the current specification of the **LoadSegBlock**. The end of data occurs when **lsb_Next** is NULL ($FFFFFFFF), and the summed data is exhausted.

```
lsb_ID                  == 'LSEG'

lsb_SummedLongs         size of this checksummed structure
                        Note that this is not 64 like most of the other
                        structures.  This is the number of valid longs
                        in this image, like bbb_SummedLongs.

lsb_ChkSum              block checksum (longword sum to zero)

lsb_HostID              SCSI Target ID of host
                        This describes the initiator ID for the creator
                        of these blocks.  (see rdb_HostID discussion)

lsb_Next                block number of the next LoadSegBlock

lsb_LoadData            data for "loadseg"
                        The data starts here and continues as long as
                        indicated by lsb_SummedLongs-5: e.g. if
                        lsb_SummedLongs is 128 (ie. for 512 byte blocks),
                        123 longs of data are valid here.
```

filesysres.h and .i

The FileSysResource is created by the first code that needs to use it. It is added to the resource list for others to use. (Checking and creation should be performed while **Forbid()** is in effect). Under Release 2 the resource is created by the system early on in the initialization sequence. Under 1.3 it is the responsibility of the first RDB driver to create it.

```
FileSysResource

  fsr_Node              on resource list with the name FileSystem.resource
  fsr_Creator           name of creator of this resource
  fsr_FileSysEntries    list of FileSysEntry structs


FileSysEntry

  fse_Node              on fsr_FileSysEntries list
                        ln_Name is of creator of this entry
  fse_DosType           DosType of this FileSys
  fse_Version           release version of this FileSys
                        Usually MSW is version, LSW is revision.

  fse_PatchFlags        bits set for those of the following that
                        need to be substituted into a standard
                        device node for this file system: e.g.
                        $180 for substitute SegList & GlobalVec

  fse_Type              device node type: zero
  fse_Task              standard dos "task" field
  fse_Lock              not used for devices: zero
  fse_Handler           filename to loadseg (if SegList is null)
  fse_StackSize         stacksize to use when starting task
  fse_Priority          task priority when starting task
  fse_Startup           startup msg: FileSysStartupMsg for disks
  fse_SegList           segment of code to run to start new task
  fse_GlobalVec         BCPL global vector when starting task

  No more entries need exist than those implied by fse_PatchFlags, so
  entries do not have a fixed size.
```

For additional information on initializing and booting a Rigid Disk Block filesystem device, see the **SCSI Device** chapter of the Addison-Wesley *Amiga ROM Kernel Reference Manual: Devices*. Writers of drivers for expansion devices that perform their own DMA (direct memory access) should consult the Exec chapters and Autodocs for information on Release 2 processor cache control functions including **CachePreDMA()** and **CachePostDMA()**. See the following include files (.h and .i) for additional notes and related structures: *<libraries/configvers>*, *<libraries/configregs>*, *<devices/hardblocks>*, *<resources/filesysres>* and *<libraries/filehandler>*.

Function Reference

The following are brief descriptions of the expansion library functions that are useful for expansion device drivers and related applications. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for the complete descriptions of all the expansion library functions.

Table 32-2: Expansion Library Functions

Function	Description
FindConfigDev()	Returns a pointer to the **ConfigDev** structure of a given expansion device.
MakeDosNode()	Creates the DOS device node for disk and similar expansion devices.
AddDosNode()	Adds a DOS device node to the system.
AddBootNode()	Adds an autobooting DOS device node to the system (V36).
GetCurrentBinding()	Returns a pointer to the **CurrentBinding** structure of a given device.
SetCurrentBinding()	Set up for reading the **CurrentBinding** with **GetCurrentBinding()**.
ObtainCurrentBinding()	Protect the **ConfigDev** structure with a semaphore.
ReleaseCurrentBinding()	Release a semaphore on **ConfigDev** set up with **ObtainCurrentBinding()**.

Chapter 33
IFFPARSE LIBRARY

The *iffparse.library* was created to help simplify the job of parsing IFF files. Unlike other IFF libraries, iffparse.library is not form-specific. This means that the job of interpreting the structure and contents of the IFF file is up to you. Previous IFF file parsers were either simple but not general, or general but not simple. IFFParse tries to be both simple *and* general.

The Structure of IFF Files

Many people have a misconception that IFF means image files. This is not the case. IFF (short for Interchange File Format) is a method of portably storing structured information in machine-readable form. The actual information can be anything, but the manner in which it is stored is very specifically detailed. This specification is the IFF standard.

The IFF standard was originally designed in 1985 by Electronic Arts in conjunction with a committee of developers. The full standard along with file descriptions and sample code is published in the *Amiga ROM Kernel Reference Manual: Devices* (3rd edition).

The goal of the IFF standard is to allow customers to move their own data between independently developed software products. The types of data objects to exchange are open-ended and currently include plain and formatted text, raster and structured graphics, fonts, music, sound effects, musical instrument descriptions, and animation. IFF addresses these needs by defining a standard for self-identifying file structures and rules for accessing these files.

CHUNKS: THE BUILDING BLOCKS OF IFF

IFF files contain various types and amounts of data grouped in data *chunks*, each starting with a four-letter ASCII identifier (the chunk ID) followed by a 32-bit length count (the chunk size). The identifier and length count make it possible for IFF readers to skip over chunks that they don't understand or don't care about, and extract only the information they need. It may be helpful to think of these chunks as the building blocks of an IFF file (Figure 33-1).

Figure 33-1: The Chunk - The Building Block of IFF

'CKID' Size
data data data data data data data data data data data ...

The 'CKID' (chunk ID) characters of a real chunk will be a four letter identifier such as 'BODY' or 'CMAP', specifying the type and format of data stored in the chunk. Most chunks contain a simple defined grouping of byte, word, and long variables similar to the contents of a C structure. Others such as sound sample and bitmap image body chunks, contain a stream of compressed data.

Chunk writing is fairly straightforward with the one caveat that all chunks must be word-aligned. Therefore an odd-length chunk must be followed by a pad byte of zero (which is not counted in the size of the chunk). When chunks are nested, the enclosing chunk must state the total size of its composite contents (including any pad bytes).

> *About Chunk Length.* Every IFF data chunk begins with a 4-character identifier field followed by a 32-bit size field (these 8 bytes are sometimes referred to as the chunk header). The size field is a count of the rest of the data bytes in the chunk, *not* including any pad byte. Hence, the total space occupied by a chunk is given by its size field (rounded to the nearest even number) + 8 bytes for the chunk header.

COMPOSITE DATA TYPES

Standard IFF files generally contain a variable number of chunks within one of the standard IFF composite data types (which may be thought of as chunks which contain other chunks). The IFF specification defines three basic composite data types, 'FORM', 'CAT ', and 'LIST'. A special composite data type, the 'PROP', is found only inside a LIST.

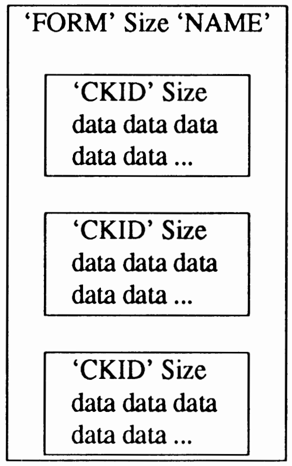
Table 33-1: Usage of Composite IFF Types

FORM	A grouping of chunks which describe one set of data
LIST	A grouping of the same type of FORMs with shared properties in a PROP
PROP	May appear in a LIST to define chunks shared by several FORMs
CAT	A concatenation of related FORMs or LISTs

The special IFF composite data types, like simple chunks, start with a 4-character identifier (such as 'FORM'), followed by a 32-bit length. But their data begins with a second 4-character identifier that tells you what type of data is in the composite. For example, a FORM ILBM contains chunks describing a bitmap image, and a FORM FTXT contains chunks describing formatted text.

It may help to think of each composite data type as a box containing chunks, the IFF building blocks. Figure 33-2 shows a diagram of a typical composite.

Figure 33-2: The FORM - The Most Common IFF File Type



The example FORM in figure 33-2 is outlined below showing the size of chunks within the FORM. Notice that the size of a composite *includes* its second 4-character identifier (shown below as 'NAME').

. FORM 72 NAME	(72 is the size of the FORM starting with the ''N'' of NAME)
. . CKID 22	(then 22 bytes of data, so chunk header + chunk size is 30)
. . CKID 10	(then 10 bytes of data, so chunk header + chunk size is 18)
. . CKID 12	(then 12 bytes of data, so chunk header + chunk size is 20)

> *Note.* In the example above, indentation represents the nesting of the chunks within the FORM. Real FORMs and chunks would have different four-character identifiers rather than 'NAME' and 'CKID':

Any odd-length chunks must have a pad byte of zero at the end of the chunk. As shown below, this pad byte is not counted in the size of the chunk but *is* counted in the size of any composite types (such as FORM) that contain an odd-length chunk. *Warning:* some IFF readers and writers do not deal with this properly.

. FORM 72 NAME	(72 is the size of the FORM starting with the ''N'' of NAME)
. . CKID 21	(then 21 bytes of data, so chunk header + chunk size is 29)
. . 0	(pad byte of zero, size 1)
. . CKID 10	(then 10 bytes of data, so chunk header + chunk size is 18)
. . CKID 12	(then 12 bytes of data, so chunk header + chunk size is 20)

Most IFF files are of the composite type FORM. Generally, a FORM is a simple grouping of chunks that provide information about some data, and the data itself. Although some standard chunks have common usage within different FORMS, the identity and format of most chunks within a FORM are relative to the definition and specification of the FORM. For example, a CRNG chunk in a FORM ILBM may have a totally different format and contents than a chunk named CRNG in a different type of FORM.

One of the most popular IFF FORMs that has been defined is the ILBM standard. This is how most Amiga bitmap imagery is stored. Since this is the most common IFF file, it is used frequently as an example.

Here is the output of a program called Sift.c that displays a text description of the contents of an IFF file (Sift.c is listed at the end of this chapter). The file shown below is a fairly simple ILBM.

```
. FORM 11068 ILBM
. . BMHD 20
. . CAMG 4
. . CMAP 12
. . BODY 10995
```

Computing the Size of a FORM. The size of the FORM (11068 bytes) is equal to the sum of the sizes stated in each chunk contained within the FORM, plus 8 bytes for the overhead of each chunk header (4 bytes for the 4-character chunk ID, and 4 bytes for the 32-bit chunk size), plus 4 bytes for the FORM's own 4-character ID ('ILBM'), plus 1 byte more for each pad byte that follow any odd-length chunks.

Parsing an IFF File

Chunk reading requires a parser to scan each chunk and dispatch the proper access/conversion procedure. For a simple IFF file, such parsing may be relatively easy, consisting mainly reading in the data of desired chunks, and seeking over unwanted chunks (and the pad byte after odd-length chunks). Interpreting nested chunks is more complex, and requires a method for keeping track of the current context, i.e., the data which is still relevant at any particular depth into the nest. The original IFF specifications compare an IFF file to a C program. Such a metaphor can be useful in understanding the scope of of the chunks in an IFF file.

The IFFParse library addresses general IFF parsing requirements by providing a run-time library which can extract the chunks you want from an IFF file, with the ability to pass control to you when it reaches a chunk that requires special processing such as decompression. IFFParse also understands complex nested IFF formats and will keep track of the context for you.

BASIC FUNCTIONS AND STRUCTURES OF IFFPARSE LIBRARY

The structures and flags of the IFFParse library are defined in the include files *<libraries/iffparse.h>* and *<libraries/iffparse.i>*. IFF files are manipulated through a structure called an **IFFHandle**. Only some of the fields in the **IFFHandle** are publicly documented. The rest are managed internally by IFFParse. This handle is passed to all IFFParse functions, and contains the current parse state and position in the file. An **IFFHandle** is obtained by calling **AllocIFF()**, and freed through **FreeIFF()**. This is the only legal way to obtain and dispose of an **IFFHandle**.

The public portion of if **IFFHandle** is defined as follows:

```
/*
 * Structure associated with an active IFF stream.
 * "iff_Stream" is a value used by the client's read/write/seek functions -
 * it will not be accessed by the library itself and can have any value
 * (could even be a pointer or a BPTR).
 */
struct IFFHandle {
        ULONG   iff_Stream;
        ULONG   iff_Flags;
        LONG    iff_Depth;      /*  Depth of context stack.  */
        /*  There are private fields hiding here.  */
};
```

Stream Management

A stream is a linear array of bytes that may be accessed sequentially or randomly. DOS files are streams. IFFParse uses Release 2 **Hook** structures (defined in <*utility/hooks.h*>) to implement a general stream management facility. This allows the IFFParse library to read, write, and seek any type of file handle or device by using an application-provided hook function to open, read, write, seek and close the stream.

Built on top of this facility, IFFParse has two internal stream managers: one for unbuffered DOS files (AmigaDOS filehandles), and one for the Clipboard.

INITIALIZATION

As shown above, the **IFFHandle** structure contains the public field **iff_Stream**. This is a longword that must be initialized to contain something meaningful to the stream manager. IFFParse never looks at this field itself (at least not directly). Your **iff_Stream** may be an AmigaDOS filehandle, or an IFFParse **ClipboardHandle**, or a custom stream of any type if you provide your own custom stream handler (see the section on "Custom Stream Handlers" later in this chapter). You must initialize your **IFFHandle** structure's **iff_Stream** to your stream, and then initialize the **IFFHandle**'s flags and stream hook.

Three sample initializations are shown here. In the case of the internal DOS stream manager, **iff_Stream** is an AmigaDOS filehandle as returned by **Open()**:

```
iff->iff_Stream = (ULONG) Open ("filename", MODE_OLDFILE);
if(iff->iff_Stream) InitIFFasDOS (iff);  /* use internal DOS stream manager */
```

In the case of the internal Clipboard stream manager, **iff_Stream** is a pointer to an IFFParse **ClipboardHandle** structure (the **OpenClipboard()** call used here is a function of iffparse.library, not the clipboard.device):

```
iff->iff_Stream = (ULONG) OpenClipboard (PRIMARY_CLIP);
InitIFFasClip (iff); /* use internal Clipboard stream manager  */
```

When using a custom handle such as an **fopen()** file handle, or a device other than the clipboard, you must provide your own flags and stream handler:

```
iff->iff_Stream = (ULONG) OpenMyCustomStream("foo");
InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
```

IFFParse "knows" the seek capabilities of DOS and **ClipboardHandle** streams, so **InitIFFasDOS()** and **InitIFFasClip()** set the flags for you.

> *You May Change the Seek Bits in iff_Flags*: IFFParse sets IFFF_FSEEK | IFFF_RSEEK for DOS files. This is not always true (e.g., pipes). If you know that a DOS file has different seek characteristics, your application may correct the seek bits in **iff_Flags** after calling **InitIFFasDOS()**.

When using **InitIFF**() to provide a custom handler, you must also provide flags to tell IFFParse the capabilities of your stream. The flags are:

IFFF_FSEEK:	This stream has forward-seek capability only.
IFFF_RSEEK:	This stream has random-seek capability. IFFF_RSEEK tends to imply IFFF_FSEEK, but it's best to specify both.

If neither flag is specified, you're telling IFFParse that it can't seek through the stream.

After your stream is initialized, call **OpenIFF**():

```
error = OpenIFF (iff, IFFF_READ);
```

Once you establish a read/write mode (by passing IFFF_READ or IFFF_WRITE), you remain in that mode until you call **CloseIFF**().

TERMINATION

Termination is simple. Just call **CloseIFF(iff)**. This may be called at any time, and terminates IFFParse's transaction with the stream. The stream itself is not closed. The **IFFHandle** may be reused; you may safely call **OpenIFF**() on it again. You are responsible for closing the streams you opened. A stream used in an **IFFHandle** must generally remain open until you **CloseIFF**().

CUSTOM STREAMS

A custom stream handler can allow you (and iffparse.library) to use your compiler's own file I/O functions such as **fopen**(), **fread**() and **fwrite**(), rather than the lower-level AmigaDOS equivalents **Open**(), **Read**(), and **Write**(). A custom stream handler could also be used to read or write IFF files from an Exec device or an unusual handler or filesystem.

If you install your own stream handler function, iffparse.library will call your function whenever it needs to read, write, or seek on your file. Your stream handler function will then perform these stream actions for iffparse.library. See the "Custom Stream Handlers" section for more information on custom stream handlers.

Parsing

This is both simple and complicated. It's simple in that it's just one call. It's complicated in that you have to seize control of the parser to get your data.

The parser operates automatically, scanning the file, verifying syntax and layout rules. If left to its default behavior, it will scan through the entire file until it reaches the end, whereupon it will tell you that it got to the end.

The whole scanning procedure is controlled through one call:

```
error = ParseIFF (iff, controlmode);
```

The control modes are IFFPARSE_SCAN, IFFPARSE_STEP and IFFPARSE_RAWSTEP. For now, only the IFFPARSE_SCAN control mode is considered.

CONTROLLING PARSING

ParseIFF(), if left to itself, wouldn't do anything useful. Ideally, it should stop at strategic places so we can look at the chunks. Here's where it can get complicated.

There are many functions provided to help control the parsing process; only the common ones are covered here. Additional functions are described in the Autodocs for iffparse.library (for the complete Autodocs, refer to the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* also by Addison-Wesley).

StopChunk()

You can instruct the parser to stop when it encounters a specific IFF chunk by using the function **StopChunk()**:

```
error = StopChunk (iff, ID_ILBM, ID_BODY);
```

When the parser encounters the requested chunk, parsing will stop, and **ParseIFF()** will return the value zero. The stream will be positioned ready to read the first data byte in the chunk. You may then call **ReadChunkBytes()** or **ReadChunkRecords()** to pull the data out of the chunk.

You may call **StopChunk()** any number of times for any number of different chunk types. If you wish to identify the chunk on which you've stopped, you may call **CurrentChunk()** to get a pointer to the current **ContextNode**, and examine the **cn_Type** and **cn_ID** fields.

Using **StopChunk()** for every chunk, you can parse IFF files in a manner very similar to the way you're probably doing it now, using a state machine. However, this would be a terrible underuse of IFFParse.

PropChunk()/FindProp()

In the case of a FORM ILBM, certain chunks are defined as being able to appear in any order. Among these are the BMHD, CMAP, and CAMG. Typically, BMHD appears first, followed by CMAP and CAMG, but you can't make this assumption. The IFF *and* ILBM standards require you to assume these chunks will appear in any order. So ideally, what you'd like to do is collect them as they arrive, but not do anything with them until you actually need them.

This is where **PropChunk()** comes in. The syntax for **PropChunk()** is identical to **StopChunk()**:

```
error = PropChunk (iff, ID_ILBM, ID_BMHD);
```

When you call **ParseIFF()**, the parser will look for chunks declared with **PropChunk()**. When it sees them, the parser will internally copy the contents of the chunk into memory for you before continuing its parsing.

When you're ready to examine the contents of the chunk, you use the function **FindProp()**:

```
StoredProperty = FindProp (iff, ID_ILBM, ID_BMHD);
```

FindProp() returns a pointer to a struct **StoredProperty**, which contains the chunk size and data. If the chunk was never encountered, NULL is returned. This permits you to process the property chunks in any order you wish, regardless of how they appeared in the file. This provides much better control of data interpretation and also reduces headaches. The following fragment shows how ILBM **BitMapHeader** data could be accessed after using **ParseIFF()** with **PropChunk**(iff, ID_ILBM, ID_BMHD):

```
struct StoredProperty *sp;     /* defined in iffparse.h */
struct BitMapHeader *bmhd;     /* defined in IFF spec   */

if (sp = FindProp(iff, ID_ILBM, ID_BMHD))
      {
        /* If property is BMHD, sp->sp_Data is ptr to data in BMHD */
        bmhd = (struct BitMapHeader *)sp->sp_Data;
        printf("BMHD: PageWidth      = %ld\n",bmhd->PageWidth);
      }
```

PUTTING IT TOGETHER

With just **StopChunk()**, **PropChunk()**, and **ParseIFF()**, you can write a viable ILBM display program. Since IFFParse knows all about IFF structure and scoping, such a display program would have the added ability to parse complex FORMs, LISTs, and CATs and attempt to find imagery buried within.

Such an ILBM reader might appear as follows:

```
iff = AllocIFF();
iff->iff_Stream = Open ("shuttle dog", MODE_OLDFILE);
InitIFFasDOS (iff);
OpenIFF (iff, IFFF_READ);

PropChunk (iff, ID_ILBM, ID_BMHD);
PropChunk (iff, ID_ILBM, ID_CMAP);
PropChunk (iff, ID_ILBM, ID_CAMG);
StopChunk (iff, ID_ILBM, ID_BODY);
ParseIFF (iff, IFFPARSE_SCAN);


if (bmhdprop = FindProp (iff, ID_ILBM, ID_BMHD))
    configurescreen (bmhdprop);
else
    bye ("No BMHD, no picture.");


if (cmapprop = FindProp (iff, ID_ILBM, ID_CMAP))
    setcolors (cmapprop);
else
    usedefaultcolors ();


if (camgprop = FindProp (iff, ID_ILBM, ID_CAMG))
    setdisplaymodes (camgprop);


decodebody (iff);
showpicture ();
CloseIFF (iff);
Close (iff->iff_Stream);
FreeIFF (iff);
```

Open the Library. Application programs must always open iffparse.library before using the functions outlined above.

Only Example Programs Skip Error Checking. Error checking is not used in the example above for the sake of clarity. A real application should always check for errors.

OTHER CHUNK MANAGEMENT FUNCTIONS

Several other functions are available for controlling the parser.

CollectionChunk() and FindCollection()

PropChunk() keeps only one copy of the declared chunk (the one currently in scope). **CollectionChunk()** collects and keeps all instances of a specified chunk. This is useful for chunks such as the ILBM CRNG chunk, which can appear multiple times in a FORM, and which don't override previous instances of themselves. **CollectionChunk()** is called identically to **PropChunk()**:

```
error = CollectionChunk (iff, type, id);
```

When you wish to find the collected chunks currently in scope, you use the function **FindCollection()**:

```
ci = FindCollection (iff, type, id);
```

You will be returned a pointer to a **CollectionItem**, which is part of a singly-linked list of all copies of the specified chunk collected so far that are currently in scope.

```
struct CollectionItem {
        struct CollectionItem   *ci_Next;
        LONG                    ci_Size;
        UBYTE                   *ci_Data;
};
```

The size of this copy of the chunk is in the **CollectionItem**'s **ci_Size** field. The **ci_Data** field points to the chunk data itself. The **ci_Next** field points to the next **CollectionItem** in the list. The last element in the list has **ci_Next** set to NULL.

The most recently-encountered instance of the chunk will be first in the list, followed by earlier chunks. Some might consider this ordering backwards.

If NULL is returned, the specified chunk was never encountered.

StopOnExit()

Whereas **StopChunk()** will stop the parser just as it enters the declared chunk, **StopOnExit()** will stop just before it leaves the chunk. This is useful for finding the end of FORMs, which would indicate that you've collected all possible data in this FORM and may now act on it.

```
/* Ask ParseIFF() to stop with IFFERR_EOC when leaving a FORM ILBM */
StopOnExit(iff,ID_ILBM, ID_FORM);
```

EntryHandler()

This is used to install your own custom chunk entry handler. See the ''Custom Chunk Handlers'' section below for more information. **StopChunk()**, **PropChunk()**, and **CollectionChunk()** are internally built on top of this.

ExitHandler()

This is used to install your own custom chunk exit handler. See the "Custom Chunk Handlers" section below for more information. **StopOnExit()** is internally built on top of this.

READING CHUNK DATA

To read data from a chunk, use the functions **ReadChunkBytes()** and **ReadChunkRecords()**. Both calls truncate attempts to read past the end of a chunk. For odd-length chunks, the parser will skip over the pad bytes for you. Remember that for chunks which have been gathered using **PropChunk()** (or **CollectionChunk()**), you may directly reference the data by using **FindProp()** (or **FindCollection()**) to get a pointer to the data. **ReadChunkBytes()** is commonly used when loading and decompressing bitmap and sound sample data or sequentially reading in data chunks such as FTXT CHRS text chunks. See the code listing ClipFTXT.c for an example usage of **ReadChunkBytes()**.

OTHER PARSING MODES

In addition to the mode IFFPARSE_SCAN, there are the modes IFFPARSE_STEP and IFFPARSE_RAWSTEP.

IFFPARSE_RAWSTEP

This mode causes the parser to progress through the stream step by step, rather than in the automated fashion provided by IFFPARSE_SCAN. In this mode, **ParseIFF()** will return upon every entry to and departure from a context.

When the parser enters a context, **ParseIFF()** will return zero. **CurrentChunk()** will report the type and ID of the chunk just entered, and the stream will be positioned to read the first byte in the chunk. When entering a FORM, LIST, CAT or PROP chunk, the longword containing the type (e.g., ILBM, FTXT, etc.) is read by the parser. In this case, the stream will be positioned to read the byte immediately following the type.)

When the parser leaves a context, **ParseIFF()** will return the value IFFERR_EOC. This is not strictly an error, but an indication that you are about to leave the current context. **CurrentChunk()** will report the type and ID of the chunk you are about to leave. The stream is not positioned predictably within the chunk.

The parser does not call any installed chunk handlers when using this mode (e.g., property chunks declared with **PropChunk()** will not be collected).

See the example program, Sift.c, for a demonstration of IFFPARSE_RAWSTEP.

IFFPARSE_STEP

This mode is identical to IFFPARSE_RAWSTEP, except that, before control returns to your code, the chunk handler (if any) for the chunk is invoked.

Writing IFF Files

IFFParse provides facilities for writing IFF files. Again, IFFParse makes no assumptions about the data you're writing, and concerns itself with verifying the syntax of your output.

CREATING CHUNKS IN A FILE

Because the IFF specification has nesting and scoping rules, you can nest chunks inside one another. One instance is the BMHD chunk, which is commonly nested inside a FORM chunk. Thus, it is necessary for you to inform IFFParse when you are starting and ending chunks.

PushChunk()

To tell IFFParse you are about to begin writing a new chunk, you use the function **PushChunk()**:

```
error = PushChunk (iff, ID_ILBM, ID_BMHD,chunksize);
```

The chunk ID and size are written to the stream. IFFParse will enforce the chunk size you specified; attempts to write past the end of the chunk will be truncated. If, as a chunk size argument, you pass IFFSIZE_UNKNOWN, the chunk will be expanded in size as you write data to it.

PopChunk()

When you are through writing data to a chunk, you complete the write by calling **PopChunk()**:

```
error = PopChunk (iff);
```

If you wrote fewer bytes than you declared with **PushChunk()**, **PopChunk()** will return an error. If you specified IFFSIZE_UNKNOWN, **PopChunk()** will seek backward on the stream and write the final size. If you specified a chunk size that was odd, **PopChunk()** will write the pad byte automatically.

PushChunk() and **PopChunk()** nest; every call to **PushChunk()** must have a corresponding call to **PopChunk()**.

WRITING CHUNK DATA

Writing the IFF chunk data is done with either the **WriteChunkBytes()** or **WriteChunkRecords()** functions.

```
error = WriteChunkBytes (iff, buf, size);
error = WriteChunkRecords (iff, buf, recsize, numrec);
```

If you specified a valid chunk size when you called **PushChunk()**, **WriteChunkBytes()** and **WriteChunkRecords()** will truncate attempts to write past the end of the chunk.

Code to write an ILBM file might take the following form:

```
iff = AllocIFF ();
iff->iff_Stream = Open ("foo", MODE_NEWFILE);
InitIFFasDOS (iff);
OpenIFF (iff, IFFF_WRITE);

PushChunk (iff, ID_ILBM, ID_FORM, IFFSIZE_UNKNOWN);

PushChunk (iff, ID_ILBM, ID_BMHD, sizeof (struct BitMapHeader));
WriteChunkBytes (iff, &bmhd, sizeof (bmhd));
PopChunk (iff);

PushChunk (iff, ID_ILBM, ID_CMAP, cmapsize);
WriteChunkBytes (iff, cmapdata, cmapsize);
PopChunk (iff);

PushChunk (iff, ID_ILBM, ID_BODY, IFFSIZE_UNKNOWN);
packwritebody (iff);
PopChunk (iff);

PopChunk (iff);

CloseIFF (iff);
Close (iff->iff_Stream);
FreeIFF (iff);
```

Again, error checking is not present for clarity. See the example code ClipFTXT.c which writes a simple FTXT clip to the clipboard.

A NOTE ON SEEKABILITY

As you can see from the above examples, IFFParse works best with a stream that can seek randomly. However, it is not possible to seek on some streams (e.g., pipes).

IFFParse will read and write streams with limited or no seek capability. In the case of reading, only forward-seek capability is desirable. Failing this, IFFParse will fake forward seeks with a number of short reads.

In the case of writing, if the stream lacks random seek capability, IFFParse will buffer *the entire contents* of the file until you do the final **PopChunk()**, or when you **CloseIFF()** the handle. At that time, the entire stream will be written in one go. This buffering happens whether or not you specify all the chunk sizes to **PushChunk()**.

> *About the Internal Buffering.* The current implementation of this internal buffering could be more efficient. Be aware that Commodore reserves the right to alter this behavior of the parser, to improve performance or reduce memory requirements. We mention this behavior on the off chance it is important to you.

Context Functions

Internally, IFFParse maintains IFF nesting and scoping context via a *context stack*. The **PushChunk()** and **PopChunk()** functions get their names from this basic idea of the iffparse.library. Direct access to this stack is not allowed. However, many functions are provided to assist in examining and manipulating the context stack.

About the Context Stack. It is probably easier to think of a stack of blocks on a table in front of you when reading this discussion.

As the nesting level increases (as would happen when parsing a nested LIST or FORM), the depth of the context stack increases; new elements are added to the top. When these contexts expire, the **ContextNodes** are deleted and the stack shrinks.

CONTEXT NODES

The current context is said to be the top element on the stack. Contextual information is stored in a structure called a **ContextNode**:

```
struct ContextNode {
        struct MinNode  cn_Node;
        LONG            cn_ID;
        LONG            cn_Type;
        LONG            cn_Size;        /*  Size of this chunk            */
        LONG            cn_Scan;        /*  # of bytes read/written so far */
        /*  There are private fields hiding here.  */
     };
```

CurrentChunk()

You can obtain a pointer to the current **ContextNode** through the function **CurrentChunk()**:

```
currentnode = CurrentChunk (iff);
```

The **ContextNode** tells you the type, ID, and size of the currently active chunk. If there is no currently active context, NULL is returned.

ParentChunk()

To find the parent of a context, you call **ParentChunk()** on the relevant **ContextNode**:

```
parentnode = ParentChunk(currentnode);
```

If there is no parent context, NULL is returned.

THE DEFAULT CONTEXT

When you first obtain an **IFFHandle** through **AllocIFF()**, a hidden default context node is created. You cannot get direct access to this node through **CurrentChunk()** or **ParentChunk()**. However, using **StoreLocalItem()**, you can store information in this context.

CONTEXT-SPECIFIC DATA: LocalContextItems

ContextNodes can contain application data specific to that context. These data objects are called **LocalContextItems**. **LocalContextItems** (**LCIs**) are a grey-box structure which contain a type, ID and identification field. **LCIs** are used to store context-sensitive data. The format of this data is application-defined. A **ContextNode** can contain as many **LCIs** as you want.

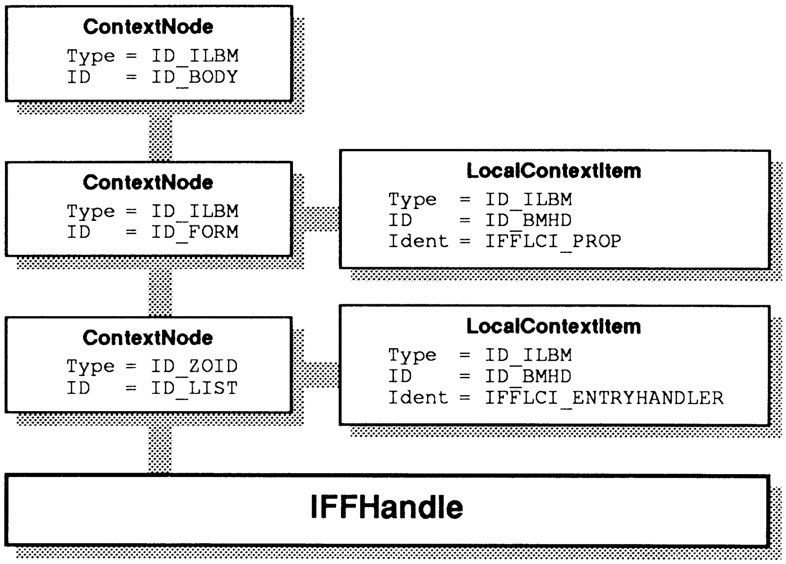


Figure 33-3: IFFParse Context Stack

Figure 33-3 shows the relationship between the **IFFHandle**, the **ContextNodes**, and the **LCIs**. The **IFFHandle** is the root of the tree, so to speak. **ContextNodes** "grow" out of the **IFFHandle**. In turn, **LCIs** "grow" out of the **ContextNodes**. What grows out of an **LCI** is client-defined.

AllocLocalItem()

To create an **LCI**, you use the function **AllocLocalItem()**:

```
lci = AllocLocalItem (type, id, ident, datasize);
```

If successful, you will be returned a pointer to an **LCI** having the specified type, ID, and identification values; and with **datasize** bytes of buffer space for your application to use.

LocalItemData()

To get a pointer to an **LCIs** data buffer, you use **LocalItemData()**:

```
buf = LocalItemData (lci);
```

You may read and write the buffer to your heart's content; it is yours. You should not, however, write beyond the end of the buffer. The size of the buffer is what you asked for when you called **AllocLocalItem()**.

STORING LCIs

Once you've created and initialized an **LCI**, you'll want to attach it to a **ContextNode**. Though a **ContextNode** can have many **LCIs**, a given **LCI** can be linked to only one **ContextNode**. Once linked, an **LCI** cannot be removed from a **ContextNode** (this may change in the future). Storing an **LCI** in a **ContextNode** is done with the functions **StoreLocalItem()** and **StoreItemInContext()**.

StoreLocalItem()

The **StoreLocalItem()** function is called as follows:

```
error = StoreLocalItem (iff, lci, position);
```

The position argument determines where the **LCI** is stored. The possible values are IFFSLI_ROOT, IFFSLI_TOP, and IFFSLI_PROP.

IFFSLI_ROOT causes **StoreLocalItem()** to store your **LCI** in the default **ContextNode**.

IFFSLI_TOP gets your **LCI** stored in the top (current) **ContextNode**.

> *The LCI Ends When the Current Context Ends.* When the current context expires, your **LCI** will be deleted by the parser.

IFFSLI_PROP causes your **LCI** to be stored in the topmost context from which a property would apply. This is usually the topmost FORM or LIST chunk. For example, suppose you had a deeply nested ILBM FORM, and you wanted to store the BMHD property in its correct context such that, when the current FORM context expired, the BMHD property would be deleted. IFFSLI_PROP will cause **StoreLocalItem()** to locate the proper context for such scoping, and store the **LCI** there. See the section on ''Finding the Prop Context'' for additional information on the scope of properties.

StoreItemInContext()

StoreItemInContext() is used when you already have a pointer to the **ContextNode** to which you want to attach your LCI. It is called like so:

```
StoreItemInContext (iff, lci, contextnode);
```

StoreItemInContext() links your **LCI** into the specified **ContextNode**. Then it searches the **ContextNode** to see if there is another **LCI** with the same type, ID, and identification values. If so, the old one is deleted.

FindLocalItem()

After you've stored your **LCI** in a **ContextNode**, you will no doubt want to be able to find it again later. You do this with the function **FindLocalItem()**, which is called as follows:

```
lci = FindLocalItem (iff, type, id, ident);
```

FindLocalItem() attempts to locate an **LCI** having the specified type, ID, and identification values. The search proceeds as follows (refer to Figure 33-3 to understand this better).

FindLocalItem() starts at the top (current) **ContextNode** and searches all **LCIs** in that context. If no matching **LCIs** are found, it proceeds down the context stack to the next **ContextNode** and searches all its **LCIs**. The process repeats until it finds the desired **LCI** (whereupon it returns a pointer to it), or reaches the end without finding anything (where it returns NULL).

> *Context Stack Position.* **LCIs** higher in the stack will ''override'' lower **LCIs** with the same type, ID, and identification field. This is how property scoping is handled. As **ContextNodes** are popped off the context stack, all its **LCIs** are deleted as well. See the section on ''Freeing LCIs'' below for additional information on deleting **LCIs**.

SOME INTERESTING INTERNAL DETAILS

> *WARNING:* This section details some internal implementation details of iffparse.library which may help you to understand it better. Use of the following information to do ''clever'' things in an application is forbidden and unsupportable. Don't even think about it.

It turns out that **StoredProperties**, **CollectionItems**, and entry and exit handlers are all implemented using **LCIs**. For example, when you call **FindProp()**, you are actually calling a front-end to **FindLocalItem()**. The mysterious identification value (which has heretofore never been discussed) is a value which permits you to differentiate between **LCIs** having the same type and ID.

For instance, suppose you called **PropChunk()**, asking it to store an ILBM BMHD. **PropChunk()** will install an entry handler in the form of an **LCI**, having type equal to 'ILBM', ID equal to 'BMHD', and an identification value of IFFLCI_ENTRYHANDLER.

When an ILBM BMHD is encountered, the entry handler is called, and it creates and stores another **LCI** having type equal to 'ILBM', ID equal to 'BMHD' and an identification value of IFFLCI_PROP.

Thus, when you call **FindProp()**, it merely calls **FindLocalItem()** with your type and ID, and supplies IFFLCI_PROP for the identification value.

Therefore, handlers, **StoredProperties**, **CollectionItems** and your own custom **LCIs** can never be confused with each other, since they all have unique identification values. Since they are all handled (and searched for) in the same way, they all ''override'' each other in a consistent way. Just as **StoredProperties** higher in the context stack will be found and returned before identical ones in lower contexts, so will chunk handlers be found and invoked before ones lower on the context stack (recall **FindLocalItem()**'s search procedure).

This means you can temporarily override a chunk handler by installing an identical handler in a higher context. The handler will persist until the context in which it is stored expires, after which, the original one regains scope.

Error Handling

If at any time during reading or writing you encounter an error, the **IFFHandle** is left in an undefined state. Upon detection of an error, you should perform an abort sequence and **CloseIFF()** the **IFFHandle**. Once **CloseIFF()** has been called, the **IFFHandle** is restored to normalcy and may be reused.

Advanced Topics

This section discusses customizing of IFFParse data handling. For applications with special needs, IFFParse supports both custom stream handlers and custom chunk handlers.

CUSTOM STREAM HANDLERS

As discussed earlier, IFFParse contains built-in stream handlers for AmigaDOS file handles as returned by **Open()**, and for the clipboard.device. If you are using AmigaDOS filehandles or the clipboard.device, you need not supply a custom stream handler.

If you wish to use your compiler's own file I/O functions (such as **fread()**) or need to read or write to an unusual handler or Exec device, you must provide a custom stream handler for your **IFFHandle**. Your custom stream handler will be called to perform all reads, writes, and seeks on your custom stream. The allows you to use compiler file I/O functions, Exec device commands, or any other method to perform the requested stream operations.

If you are implementing your own custom stream handler, you will need to know the mechanics of hook call-backs, and how to interpret the parameters. An IFFParse custom stream handler is simply a function in your code that follows Release 2 hook function conventions (Hook functions are also known as callback functions. See the ''Utility Library'' chapter for more details).

Installing a Custom Stream Handler

To initialize your **IFFHandle** to point to your custom stream and stream handler, you must open your stream and place a pointer to the stream in your **IFFHandle**'s **iff_Stream** field. This pointer could be the return from **fopen()**, or an Exec device I/O request, or any other type of pointer which will provide your custom stream handler with a way to manage your stream. For example:

```
iff->iff_Stream = (ULONG) fopen("foo");
```

Next, you must install your custom handler for this stream, and also tell IFFParse the seek capabilities of your stream. To install your custom stream handler, you must first set up a **Hook** structure (*<utility/hooks.h>*) to point to your stream handling function. Then use **InitIFF()** to install your stream handler and also tell IFFParse the seek capabilities of your stream. There is ''some assembly required''.

Release 2 hook function calling conventions specify a register interface. For an IFFParse custom stream hook, at entry, A0 contains a pointer to the hook, A2 is a pointer to your **IFFHandle**, and A1 is a pointer to a command packet, which tells you what to do. A6 contains a pointer to **IFFParseBase**. You may trash A0, A1, D0, and D1. All other registers *must* be preserved. You return your error code in D0. A return of 0 indicates success. A non-zero return indicates an error.

If your compiler supports registered function parameters, you may use a registerized C function entry as the **h_Entry** hook entry point:

```
/* mystreamhandler - SAS C custom stream handler with registerized arguments */
static LONG __saveds __asm
mystreamhandler (
    register __a0 struct Hook           *hook,
    register __a2 struct IFFHandle      *iff,
    register __a1 struct IFFStreamCmd   *actionpkt
    )
{
/*
 * Code to handle the stream commands - see end this section
 * for a complete example custom stream handler function.
 *
 */
}


/* Initialization of Hook structure for registerized handler function */
struct Hook mystreamhook {
      { NULL },
      (ULONG (*)()) mystreamhandler,      /* h_Entry, registerized function entry */
      NULL,
      NULL };


/* Open custom stream and InitIFF to custom stream handler */
if ( iff->iff_Stream = (ULONG) myopen("foo") )
      {
      InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
      }
```

Alternately, you could initialize your **Hook** structure's **h_Entry** to point to a standard assembler stub which would push the register arguments on the stack and then call a standard args C function. In this case, you must store the address of your C function in the **h_SubEntry** field of the **Hook** structure so the assembler stub can find and call your C entry point. A sample assembler hook entry stub follows. This would be assembled as hookentry.o and linked with your C code.

```
* HookEntry.asm for SAS C

      INCLUDE "exec/types.i"
      INCLUDE "utility/hooks.i"

      XDEF    _HookEntry

      section code

 _HookEntry:
      move.l  a6,-(sp)
      move.l  a1,-(sp)                ; push message packet pointer
      move.l  a2,-(sp)                ; push object pointer
      move.l  a0,-(sp)                ; push hook pointer
      move.l  h_SubEntry(a0),a0       ; fetch C entry point ...
      jsr     (a0)                    ; ... and call it
      lea     12(sp),sp               ; fix stack
      move.l  (sp)+,a6
      rts

      end
```

When using an assembler **HookEntry** stub, your C program's custom stream handler interface would be initialized as follows:

```
extern LONG HookEntry();        /* The assembler entry */

/* mystreamhandler - a standard args C function custom stream handler */
static LONG mystreamhandler ( struct Hook *hook,
                              struct IFFHandle *iff,
                              struct IFFStreamCmd *actionpkt )
{
/* Code to handle the stream commands - see end of this section
 * for a complete example custom stream handler function.
 */
}

/* Initialization of Hook for asm HookEntry and std args C function */
struct Hook mystreamhook {
      { NULL },
      (ULONG (*)()) HookEntry,      /* h_Entry, assembler stub entry */
      (ULONG (*)()) mystreamhandler,      /* h_SubEntry, std args function entry */
      NULL };

/* Open custom stream and InitIFF to custom stream handler */
if ( iff->iff_Stream = (ULONG) myopen("foo") )
      {
      InitIFF (iff, IFFF_FSEEK | IFFF_RSEEK, &mystreamhook);
      }
```

Inside a Custom Stream Handler

When the library calls your stream handler, you'll be passed a pointer to the **Hook** structure (**hook** in the example used here), a pointer to the **IFFHandle** (**iff**), and a pointer to an **IFFStreamCmd** structure (**actionpkt**). Your stream pointer may be found in **iff->iff_Stream** where you previously stored it. The **IFFStreamCmd** (**actionpkt**) will describe the action that IFFParse needs you to perform on your stream:

```
/* Custom stream handler is passed struct IFFStreamCmd *actionpkt */
struct IFFStreamCmd {
        LONG    sc_Command;     /* Operation to be performed (IFFCMD_) */
        APTR    sc_Buf;         /* Pointer to data buffer              */
        LONG    sc_NBytes;      /* Number of bytes to be affected      */
      };

/* Possible call-back command values.  (Using 0 as the value for IFFCMD_INIT
 * was, in retrospect, probably a bad idea.)
 */
#define IFFCMD_INIT      0      /* Prepare the stream for a session    */
#define IFFCMD_CLEANUP   1      /* Terminate stream session            */
#define IFFCMD_READ      2      /* Read bytes from stream              */
#define IFFCMD_WRITE     3      /* Write bytes to stream               */
#define IFFCMD_SEEK      4      /* Seek on stream                      */
#define IFFCMD_ENTRY     5      /* You just entered a new context      */
#define IFFCMD_EXIT      6      /* You're about to leave a context     */
#define IFFCMD_PURGELCI  7      /* Purge a LocalContextItem            */
```

Your custom stream handler should perform the requested action on your custom stream, and then return 0 for success or an IFFParse error if an error occurred. The following code demonstrates a sample stream handler for a stream which was opened with a compiler's **fopen()** buffered file I/O function:

```
static LONG mystreamhandler ( struct Hook *hook,
                              struct IFFHandle *iff,
                              struct IFFStreamCmd *actionpkt )
{
     register FILE     *stream;
     register LONG     nbytes, error;
     register UBYTE    *buf;
```

```
    stream      = (FILE *) iff->iff_Stream;   /* get your stream pointer */
    nbytes      = actionpkt->sc_NBytes;       /* length for command */
    buf   = (UBYTE *) actionpkt->sc_Buf;      /* buffer for the command */

    /* Now perform the action specified by the actionpkt->sc_Command */

    switch (actionpkt->sc_Command) {
    case IFFCMD_READ:
          /*
           * IFFCMD_READ means read sc_NBytes from the stream and place
           * it in the memory pointed to by sc_Buf.  Be aware that
           * sc_NBytes may be larger than can be contained in an int.
           * This is important if you plan on recompiling this for
           * 16-bit ints, since fread() takes int arguments.
           *
           * Any error code returned will be remapped by IFFParse into
           * IFFERR_READ.
           */
          error = (fread (buf, 1, nbytes, stream) != nbytes);
          break;

    case IFFCMD_WRITE:
          /*
           * IFFCMD_WRITE is analogous to IFFCMD_READ.
           *
           * Any error code returned will be remapped by IFFParse into
           * IFFERR_WRITE.
           */
          error = (fwrite (buf, 1, nbytes, stream) != nbytes);
          break;

    case IFFCMD_SEEK:
          /*
           * IFFCMD_SEEK asks that you performs a seek relative to the
           * current position.  sc_NBytes is a signed number,
           * indicating seek direction (positive for forward, negative
           * for backward).  sc_Buf has no meaning here.
           *
           * Any error code returned will be remapped by IFFParse into
           * IFFERR_SEEK.
           */
          error = (fseek (stream, nbytes, 1) == -1);
          break;

    case IFFCMD_INIT:
          /*
           * IFFCMD_INIT means to prepare your stream for reading.
           * This is used for certain streams that can't be read
           * immediately upon opening, and need further preparation.
           * This operation is allowed to fail;  the error code placed
           * in D0 will be returned directly to the client.
           *
           * An example of such a stream is the clipboard.  The
           * clipboard.device requires you to set the io_ClipID and
           * io_Offset to zero before starting a read.  You would
           * perform such a sequence here.  (Stdio files need no such
           * preparation, so we simply return zero for success.)
           */
    case IFFCMD_CLEANUP:
          /*
           * IFFCMD_CLEANUP means to terminate the transaction with
           * the associated stream.  This is used for streams that
           * can't simply be closed.  This operation is not allowed to
           * fail;  any error returned will be ignored.
           *
           * An example of such a stream is (surprise!) the clipboard.
           * It requires you to explicitly end reads by CMD_READing
           * past the end of a clip, and end writes by sending a
           * CMD_UPDATE.  You would perform such a sequence here.
           * (Again, stdio needs no such sequence.)
           */
          error = 0;
          break;
    }
    return (error);
}
```

CUSTOM CHUNK HANDLERS

Like custom stream handlers, custom chunk handlers are implemented using Release 2 **Hook** structures. See the previous section for details on how a handler function may be interfaced using a **Hook** structure.

There are two types of chunk handlers: entry handlers and exit handlers. Entry handlers are invoked just after the parser enters the chunk; the stream will be positioned to read the first byte in the chunk. (If the chunk is a FORM, LIST, CAT, or PROP, the longword type will be read by the parser; the stream will be positioned to read the byte immediately following the type.) Exit handlers are invoked just before the parser leaves the chunk; the stream is not positioned predictably within the chunk.

Installing a Custom Chunk Handler

To install an entry handler, you call **EntryHandler()** in the following manner:

```
error = EntryHandler (iff, type, id, position, hookptr, object);
```

An exit handler is installed by saying:

```
error = ExitHandler (iff, type, id, position, hookptr, object);
```

In both cases, a handler is installed for chunks having the specified **type** and **id**. The **position** argument specifies in what context to install the handler, and is identical to the position argument used by **StoreLocalItem()**. The **hookptr** argument given above is a pointer to your **Hook** structure.

Inside a Custom Chunk Handler

The mechanics of receiving parameters through the **Hook** are covered in the "Custom Stream Handlers" section, and are not duplicated here. Refer to the **EntryHandler()** and **ExitHandler()** Autodocs for the contents of the registers upon entry.

Once inside your handler, you can call nearly all functions in the iffparse.library, however, you should avoid calling **ParseIFF()** from within chunk handlers.

Your handler runs in the same environment as whoever called **ParseIFF()**. The propagation sequence is:

[mainline] --calls--> [ParseIFF()] --calls--> [your_handler()]

Thus, your handler runs on your mainline's stack, and can call any OS functions the mainline code can. (Your handler will have to set the global base pointer if your code uses base-relative addressing.)

When leaving the handler, you must follow the standard register preservation conventions (D0/D1/A0/A1 may be trashed, all others must be preserved). D0 contains your return code, which will affect the parser in a number of ways:

If you return zero (a normal, uneventful return), **ParseIFF()** will continue normally. If you return the value IFFERR_RETURN2CLIENT, **ParseIFF()** will stop and return the value zero to the mainline code.

If you return any other value, **ParseIFF()** will stop and return that value to the mainline code. This is how you should return error conditions to the client code.

The Object Parameter

The **object** parameter supplied to **EntryHandler()** and **ExitHandler()** is a pointer which will be passed to your handler when invoked. This pointer can be anything; the parser doesn't use it directly. As an example, you might pass the pointer to the **IFFHandle**. This way, when your handler is called, you'll be able to easily perform operations on the **IFFHandle** within your handler. Such code might appear as follows:

```
error = EntryHandler (iff, ID_ILBM, ID_BMHD, IFFSLI_ROOT, hook, iff);
```

And your handler would be declared as follows:

```
/* Registerized handler hook h_Entry using SAS C registerized parameters */
LONG __asm MyHandler (register __a0 struct Hook      *hook,
                      register __a1 LONG             *cmd,
                      register __a2 struct IFFHandle *iff)

/* Standard args handler hook h_SubEntry using HookEntry.asm as Hook h_Entry */
LONG MyHandler ( struct Hook      *hook,
                 LONG             *cmd,
                 struct IFFHandle *iff )
```

From within your handler, you could then call **CurrentChunk()**, **ReadChunkBytes()**, or nearly any other operation on the **IFFHandle**. Please refer to the **EntryHandler()** and **ExitHandler()** Autodocs for additional information on the use of chunk handlers.

FINDING THE PROP CONTEXT

Earlier it was mentioned that supplying a position value of IFFSLI_PROP to **StoreLocalItem()** would store it in the topmost property scope. **FindPropContext()** is the routine that finds that topmost context.

Property chunks (such as the BMHD, CMAP, and others) have dominion over the FORM that contains them; they are said to be ''in scope'' and their definition persists until the FORM's context ends. Thus, a property chunk has a scoping level equal to the FORM that contains it; when the FORM ends, the property dies with it.

Consider a more complicated example. Suppose you have a LIST with a PROP in it. PROPs are the global variables of LISTs; thus a property chunk declared in a PROP will persist until the LIST's context ends.

This is what **FindPropContext()** is looking for; a context level in which a property chunk may be installed.

FindPropContext() starts at the parent of the current context (second from the top of the context stack) and starts searching downward, looking for the first FORM or LIST context it finds. If it finds one, it returns a pointer to that **ContextNode**. If it can't find a suitable context level, it returns NULL.

FindPropContext() is called as follows:

```
struct ContextNode *cn;

cn = FindPropContext (iff);
```

FREEING LCIs

Ordinarily, the parser will automatically delete **LCIs** you have allocated and installed. However, you may have a case where simply **FreeMem()**ing your **LCI** is not enough; you may need to free some ancillary memory, or decrement a counter, or send a signal, or something. This is where **SetLocalItemPurge()** comes in. It is called as follows:

```
SetLocalItemPurge (lci, hookptr);
```

When the parser is ready to delete your **LCI**, your purge handler code will be called through the **Hook** you supplied. You can then perform all your necessary operations. One of these operations should be to free the **LCI** itself. This is done with **FreeLocalItem()**:

```
FreeLocalItem (lci);
```

This deallocates the memory used to store the **LCI** and the client buffer allocated with it. **FreeLocalItem()** is only called as part of a custom purge handler.

As with custom chunk handlers, your purge handler executes in the same environment as the mainline code that called **ParseIFF()**. It is recommended that you keep purge handlers short and to the point; super clever stuff should be reserved for custom chunk handlers, or for the client's mainline code. Custom purge handlers must *always* work; failures will be ignored.

IFF FORM Specifications

The specifications for Amiga IFF formats are maintained by Commodore Applications and Technical Support (CATS) and updated periodically. The latest specifications are published in the *Amiga ROM Kernel Reference Manual: Devices* (3rd edition) and also available in electronic form directly from CATS. Between updates of the IFF Manual, selected new FORMs and changes to existing FORMs are documented in *Amiga Mail* a technical newsletter for Amiga developers published by Commodore's CATS group.

Some of the most commonly used IFF FORMs are the four that were originally specified in the EA IFF-85 standard:

ILBM	Bitmap images and palettes
FTXT	Simple formatted text
SMUS	Simple musical scores
8SVX	8-bit sound samples

Of these four, ILBM is the most commonly encountered FORM, and FTXT is becoming increasingly important since the Release 2 *conclip* command passes clipped console text through the clipboard as FTXT. All data clipped to the clipboard must be in an IFF format.

This section will provide a brief summary of the ILBM and FTXT FORMs and their most used common chunks. Please consult the EA-IFF specifications for additional information.

FORM ILBM

The IFF file format for graphic images on the Amiga is called FORM ILBM (InterLeaved BitMap). It follows a standard parsable IFF format.

ILBM.BMHD BitMapHeader Chunk

The most important chunk in a FORM ILBM is the BMHD (BitMapHeader) chunk which describes the size and compression of the image:

```
typedef UBYTE Masking;  /* Choice of masking technique - Usually 0. */
#define mskNone                 0
#define mskHasMask              1
#define mskHasTransparentColor  2
#define mskLasso                3

/* Compression algorithm applied to the rows of all source
 * and mask planes. "cmpByteRun1" is byte run encoding.
 * Do not compress across rows!  Compression is usually 1.
 */
typedef UBYTE Compression;
#define cmpNone                 0
#define cmpByteRun1             1

/* The BitMapHeader structure expressed as a C structure */
typedef struct {
        UWORD w, h;             /* raster width & height in pixels      */
        WORD  x, y;             /* pixel position for this image        */
        UBYTE nPlanes;          /* # bitplanes (without mask, if any)   */
        Masking     masking;  /* One of the values above.  Usually 0  */
        Compression compression;/* One of the values above.  Usually 1  */
        UBYTE reserved1;      /* reserved; ignore on read, write as 0 */
        UWORD transparentColor; /* transparent color number. Usually 0  */
        UBYTE xAspect, yAspect; /* pixel aspect, a ratio width : height */
        WORD  pageWidth, pageHeight;    /* source "page" size in pixels */
} BitMapHeader;
```

Sample Hex Dump of an ILBM

WARNING: This hex dump is shown only to help the reader understand how IFF chunks appear in a file. You cannot ever depend on any particular ILBM chunk being at any particular offset into the file. IFF files are composed, in their simplest form, of chunks within a FORM. Each chunk starts with a 4-letter chunkID, followed by a 32-bit length of the rest of the chunk. You must *parse* IFF files, skipping past unneeded or unknown chunks by seeking their length (+1 if odd length) to the next 4-letter chunk ID. In a real ILBM file, you are likely to encounter additional optional chunks. See the IFF Specification listed in the *Amiga ROM Kernel Reference Manual: Devices* for additional information on such chunks.

```
0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD
0010: 00000014 01400190 00000000 06000100    .....@..........
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....
0030: 00000804 434D4150 00000030 001000E0    ....CMAP...0....
0040: E0E00000 20000050 30303050 50500030    .... ..P000PPP.0
0050: 90805040 70707010 60E02060 E06080D0    ..P@ppp.`. `.`..
0060: A0A0A0A0 90E0C0C0 C0D0A0E0 424F4459    ............BODY
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...  etc.

Interpretation:

      'F O R M' length  'I L B M''B M H D'<-start of BitMapHeader chunk
0000: 464F524D 00016418 494C424D 424D4844    FORM..d.ILBMBMHD

       length  WideHigh XorgYorg PlMkCoRe <- Planes Mask Compression Reserved
0010: 00000014 01400190 00000000 06000100    .....@..........
```

```
      TranAspt PagwPagh 'C A M G' length  <- start of C-AMiGa View modes chunk
0020: 00000A0B 01400190 43414D47 00000004    .....@..CAMG....

dir include:
      ViewMode 'C M A P' length   R g b R <- ViewMode 800=HAM | 4=LACE
0030: 00000804 434D4150 00000030 001000E0    ....CMAP...0....

       g b R g  b R g b  R g b R  g b R g <- Rgb's are for reg0 thru regN
0040: E0E00000 20000050 30303050 50500030    .... ..P000PPP.0

       b R g b  R g b R  g b R g  b R g b
0050: 90805040 70707010 60E02060 E06080D0    ..P@ppp.`. `.`..

       R g b R  g b R g  b R g b 'B O D Y'
0060: A0A0A0A0 90E0C0C0 C0D0A000 424F4459    ............BODY

       length   start of body data        <- Compacted (Compression=1 above)
0070: 000163AC F8000F80 148A5544 2ABDEFFF    ..c.......UD*...
0080: FFBFF800 0F7FF7FC FF04F85A 77AD5DFE    ............Zw.].  etc.

Simple CAMG ViewModes:  HIRES=0x8000  LACE=0x4  HAM=0x800  HALFBRITE=0x80

( Release 2 ILBMs may contain a LONGWORD ViewPort ModeID in CAMG )
```

Interpreting ILBMs

ILBM is a fairly simple IFF FORM. All you really need to deal with to extract the image are the following chunks:

BMHD
: Information about the size, depth, compaction method (see interpreted hex dump above).

CAMG
: Optional Amiga ViewModes chunk. Most HAM and HALFBRITE ILBMs should have this chunk. If no CAMG chunk is present, and image is 6 planes deep, assume HAM and you'll probably be right. Some Amiga ViewModes flags are HIRES=0x8000, LACE=0x4, HAM=0x800, HALFBRITE=0x80. 2.0 ILBM writers should write a full 32-bit mode ID in the CAMG. See the IFF Manual for more information on writing and interpreting 32-bit CAMG values.

CMAP
: RGB values for color registers 0 to N. Previously, 4-bit RGB components each left justified in a byte. These should now be stored as a full 8-bit RGB values by duplicating 4-bit values in the high and low nibble of the byte.

BODY
: The pixel data, stored in an interleaved fashion as follows (each line individually compacted if BMHD Compression = 1):

plane 0 scan line 0
plane 1 scan line 0
plane 2 scan line 0
plane n scan line 0
plane 0 scan line 1
plane 1 scan line 1
etc.

Optional Chunks
: Also watch for AUTH Author chunks and (c) copyright chunks and preserve any copyright information if you rewrite the ILBM.

ILBM BODY Compression

The BODY contains pixel data for the image. Width, Height, and depth (Planes) is specified in the BMHD.

If the BMHD Compression byte is 0, then the scan line data is not compressed. If Compression=1, then each scan line is individually compressed as follows:

```
while (not produced the desired number of bytes)

        /* get a byte, call it N */

        if (N >= 0 && N <= 127)
                /* copy the next N+1 bytes literally */

        if (N >= -127 && N <= -1)
                /* repeat the next byte N+1 times */

        if (N == -128)
                /* skip it, presumably it's padding */
```

Interpreting the Scan Line Data

If the ILBM is not HAM or HALFBRITE, then after parsing and uncompacting if necessary, you will have N planes of pixel data. Color register used for each pixel is specified by looking at each pixel thru the planes. For instance, if you have 5 planes, and the bit for a particular pixel is set in planes 0 and 3:

```
PLANE     4 3 2 1 0
PIXEL     0 1 0 0 1
```

then that pixel uses color register binary 01001 = 9.

The RGB value for each color register is stored in the CMAP chunk of the ILBM, starting with register 0, with each register's RGB value stored as one byte of R, one byte G, and one byte of B, with each component left justified in the byte. (ie. Amiga R, G, and B components are each stored in the high nibble of a byte)

But, if the picture is HAM or HALFBRITE, it is interpreted differently. Hopefully, if the picture is HAM or HALFBRITE, the package that saved it properly saved a CAMG chunk (look at a hex dump of your file with ASCII interpretation - you will see the chunks - they all start with a 4-ASCII-char chunk ID). If the picture is 6 planes deep and has no CAMG chunk, it is probably HAM. If you see a CAMG chunk, the 'CAMG' is followed by the 32-bit chunk length, and then the 32-bit Amiga view mode flags.

HAM pictures will have the 0x800 bit set in CAMG chunk ViewModes. HALBRITE pictures will have the 0x80 bit set. See the graphics library chapters or the *Amiga Hardware Reference Manual* for more information on HAM and HALFBRITE modes.

Other ILBM Notes

Amiga ILBMs images must be stored as an even number of bytes in width. However, the ILBM BMHD field w (width) should describe the actual image width, not the rounded up width as stored.

ILBMs created with Electronic Arts IBM or Amiga *Deluxe Paint II* packages are compatible (though you may have to use a '.lbm' filename extension on an IBM). The ILBM graphic files may be transferred between the machines (or between the Amiga and IBM sides your Amiga if you have a CBM Bridgeboard card installed) and loaded into either package.

FORM FTXT

The FTXT (Formatted TeXT) form is the standard format for sharing text on the Amiga. The Release 2 console device clip and paste functions, in conjunction with the startup-sequence *conclip* command, pass clipped console text through the clipboard as FTXT.

By supporting reading and writing of clipboard device FTXT, your application will allow users to cut and paste text between your application, other applications, and Amiga Shell windows.

The FTXT form is very simple. Generally, for clip and paste operations, you will only be concerned with the CHRS (character string) chunks of an FTXT. There may be one or more CHRS chunks, each of which contains a non-NULL-terminated string of ASCII characters whose length is equal to the length (**StoredProperty.sp_Size**) of the chunk.

Be aware that if you **CollectionChunk()** the CHRS chunks of an FTXT, the list accumulated by IFFParse will be in reverse order from the chunk order in the file. See the ClipFTXT.c example at the end of this chapter for a simple example of reading (and optionally writing) FTXT to and from the clipboard. See also the ''Clipboard Device'' and ''Console Device'' chapters of the *Amiga ROM Kernel Reference Manual: Devices* for more information on Release 2 console clip and paste.

IFFParse Examples

Two examples follow: ClipFTXT.c, and Sift.c. These are simple examples that demonstrate the use of the IFFParse library. More complex examples for displaying and saving ILBMs may be found in the IFF Appendix of the *Amiga ROM Kernel Reference Manual: Devices.*

CLIPBOARD FTXT EXAMPLE

ClipFTXT.c demonstrates reading (and optional writing) of clipboard device FTXT. This example may be used as the basis for supporting Release 2 console pastes.

```
;/* clipftxt.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -j73 clipftxt.c
Blink FROM LIB:c.o,clipftxt.o TO clipftxt LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

*
* clipftxt.c:     Writes ASCII text to clipboard unit as FTXT
*          (All clipboard data must be IFF)
*
* Usage: clipftxt unitnumber
*
* To convert to an example of reading only, comment out #define WRITEREAD
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
```

```
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* Causes example to write FTXT first, then read it back
 * Comment out to create a reader only
 */
#define WRITEREAD


#define MINARGS 2

/* 2.0 Version string for c:Version to find */
UBYTE vers[] = "\0$VER: clipftxt 37.2";

UBYTE usage[] = "Usage: clipftxt unitnumber (use zero for primary unit)";

/*
 * Text error messages for possible IFFERR_#? returns from various
 * IFF routines.  To get the index into this array, take your IFFERR code,
 * negate it, and subtract one.
 *  idx = -error - 1;
 */
char  *errormsgs[] = {
      "End of file (not an error).",
      "End of context (not an error).",
      "No lexical scope.",
      "Insufficient memory.",
      "Stream read error.",
      "Stream write error.",
      "Stream seek error.",
      "File is corrupt.",
      "IFF syntax error.",
      "Not an IFF file.",
      "Required call-back hook missing.",
      "Return to client.  You should never see this."
};

#define RBUFSZ 512

#define  ID_FTXT  MAKE_ID('F','T','X','T')
#define  ID_CHRS  MAKE_ID('C','H','R','S')

struct Library *IFFParseBase;

UBYTE mytext[]="This FTXT written to clipboard by clipftxt example.\n";

void main(int argc, char **argv)
{
    struct IFFHandle    *iff = NULL;
    struct ContextNode  *cn;
    long          error=0, unitnumber=0, rlen;
    int textlen;
    UBYTE readbuf[RBUFSZ];

      /* if not enough args or '?', print usage */
      if(((argc)&&(argc<MINARGS))||(argv[argc-1][0]=='?'))
            {
            printf("%s\n",usage);
            exit(RETURN_WARN);
            }

      unitnumber = atoi(argv[1]);

      if (!(IFFParseBase = OpenLibrary ("iffparse.library", 0L)))
            {
            puts("Can't open iff parsing library.");
            goto bye;
            }

      /*
       * Allocate IFF_File structure.
       */
      if (!(iff = AllocIFF ()))
            {
            puts ("AllocIFF() failed.");
```

```
        goto bye;
        }

    /*
     * Set up IFF_File for Clipboard I/O.
     */
    if (!(iff->iff_Stream = (ULONG) OpenClipboard (unitnumber)))
        {
        puts ("Clipboard open failed.");
        goto bye;
        }
    else printf("Opened clipboard unit %ld\n",unitnumber);

    InitIFFasClip (iff);

#ifdef WRITEREAD

    /*
     * Start the IFF transaction.
     */
    if (error = OpenIFF (iff, IFFF_WRITE))
        {
        puts ("OpenIFF for write failed.");
        goto bye;
        }

    /*
     * Write our text to the clipboard as CHRS chunk in FORM FTXT
     *
     * First, write the FORM ID (FTXT)
     */
    if(!(error=PushChunk(iff, ID_FTXT, ID_FORM, IFFSIZE_UNKNOWN)))
        {
        /* Now the CHRS chunk ID followed by the chunk data
         * We'll just write one CHRS chunk.
         * You could write more chunks.
         */
        if(!(error=PushChunk(iff, 0, ID_CHRS, IFFSIZE_UNKNOWN)))
            {
            /* Now the actual data (the text) */
            textlen = strlen(mytext);
            if(WriteChunkBytes(iff, mytext, textlen) != textlen)
                {
                puts("Error writing CHRS data.");
                error = IFFERR_WRITE;
                }
            }
        if(!error) error = PopChunk(iff);
        }
    if(!error) error = PopChunk(iff);


    if(error)
        {
        printf ("IFF write failed, error %ld: %s\n",
            error, errormsgs[-error - 1]);
        goto bye;
        }
    else printf("Wrote text to clipboard as FTXT\n");

    /*
     * Now let's close it, then read it back
     * First close the write handle, then close the clipboard
     */
    CloseIFF(iff);
    if (iff->iff_Stream) CloseClipboard ((struct ClipboardHandle *)
                                iff->iff_Stream);

    if (!(iff->iff_Stream = (ULONG) OpenClipboard (unitnumber)))
        {
        puts ("Reopen of Clipboard failed.");
        goto bye;
        }
    else printf("Reopened clipboard unit %ld\n",unitnumber);

#endif /* WRITEREAD */
```

```
    if (error = OpenIFF (iff, IFFF_READ))
        {
        puts ("OpenIFF for read failed.");
        goto bye;
        }

    /* Tell iffparse we want to stop on FTXT CHRS chunks */
    if (error = StopChunk(iff, ID_FTXT, ID_CHRS))
        {
        puts ("StopChunk failed.");
        goto bye;
        }

    /* Find all of the FTXT CHRS chunks */
    while(1)
        {
        error = ParseIFF(iff,IFFPARSE_SCAN);
        if(error == IFFERR_EOC) continue;   /* enter next context */
        else if(error) break;

        /* We only asked to stop at FTXT CHRS chunks
         * If no error we've hit a stop chunk
         * Read the CHRS chunk data
         */
        cn = CurrentChunk(iff);

        if((cn)&&(cn->cn_Type == ID_FTXT)&&(cn->cn_ID == ID_CHRS))
            {
            printf("CHRS chunk contains:\n");
            while((rlen = ReadChunkBytes(iff,readbuf,RBUFSZ)) > 0)
                {
                Write(Output(),readbuf,rlen);
                }
            if(rlen < 0)      error = rlen;
            }
        }

    if((error)&&(error != IFFERR_EOF))
        {
        printf ("IFF read failed, error %ld: %s\n",
            error, errormsgs[-error - 1]);
        }

bye:
    if (iff) {
        /*
         * Terminate the IFF transaction with the stream.  Free
         * all associated structures.
         */
        CloseIFF (iff);

        /*
         * Close the clipboard stream
         */
        if (iff->iff_Stream)
                CloseClipboard ((struct ClipboardHandle *)
                        iff->iff_Stream);
        /*
         * Free the IFF_File structure itself.
         */
        FreeIFF (iff);
        }
    if (IFFParseBase) CloseLibrary (IFFParseBase);

    exit (RETURN_OK);
}
```

IFF SCANNER EXAMPLE

Sift.c lists the type and size of every chunk in an IFF file and, and checks the IFF file for correct syntax. You should use Sift to check IFF files created by your programs.

```
;/* sift.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -j73 sift.c
Blink FROM LIB:c.o,sift.o TO sift LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

*
* sift.c:   Takes any IFF file and tells you what's in it.  Verifies syntax and all that cool stuff.
*
* Usage: sift -c       ; For clipboard scanning
*    or  sift <file>         ; For DOS file scanning
*
* Reads the specified stream and prints an IFFCheck-like listing of the contents of the IFF file, if any.
* Stream is a DOS file for <file> argument, or is the clipboard's primary clip for -c.
* This program must be run from a CLI.
*
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/iffparse.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/iffparse_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }     /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

#define MINARGS 2

UBYTE vers[] = "\0$VER: sift 37.1";       /* 2.0 Version string for c:Version to find */
UBYTE usage[] = "Usage: sift IFFfilename (or -c for clipboard)";

void PrintTopChunk (struct IFFHandle *);  /* proto for our function */

/*
 * Text error messages for possible IFFERR_#? returns from various IFF routines.  To get the index into
 * this array, take your IFFERR code, negate it, and subtract one.
 *  idx = -error - 1;
 */
char  *errormsgs[] = {
      "End of file (not an error).", "End of context (not an error).", "No lexical scope.",
      "Insufficient memory.", "Stream read error.", "Stream write error.",
      "Stream seek error.", "File is corrupt.", "IFF syntax error.",
      "Not an IFF file.", "Required call-back hook missing.", "Return to client.  You should never see this."
};

struct Library *IFFParseBase;

void main(int argc, char **argv)
{
    struct IFFHandle    *iff = NULL;
    long            error;
    short           cbio;

      /* if not enough args or '?', print usage */
      if(((argc)&&(argc<MINARGS))||(argv[argc-1][0]=='?'))
            {
            printf("%s\n",usage);
            goto bye;
            }

      /* Check to see if we are doing I/O to the Clipboard. */
      cbio = (argv[1][0] == '-'  &&  argv[1][1] == 'c');
```

```
if (!(IFFParseBase = OpenLibrary ("iffparse.library", 0L)))
      {
      puts("Can't open iff parsing library.");
      goto bye;
      }

/* Allocate IFF_File structure. */
if (!(iff = AllocIFF ()))
      {
      puts ("AllocIFF() failed.");
      goto bye;
      }

/*
 * Internal support is provided for both AmigaDOS files, and the clipboard.device.  This bizarre
 * 'if' statement performs the appropriate machinations for each case.
 */
if (cbio)
      {
      /*
       * Set up IFF_File for Clipboard I/O.
       */
      if (!(iff->iff_Stream =
                  (ULONG) OpenClipboard (PRIMARY_CLIP)))
            {
            puts ("Clipboard open failed.");
            goto bye;
            }
      InitIFFasClip (iff);
      }
else
      {
      /* Set up IFF_File for AmigaDOS I/O.  */
      if (!(iff->iff_Stream = Open (argv[1], MODE_OLDFILE)))
            {
            puts ("File open failed.");
            goto bye;
            }
      InitIFFasDOS (iff);
      }

/* Start the IFF transaction. */
if (error = OpenIFF (iff, IFFF_READ))
      {
      puts ("OpenIFF failed.");
      goto bye;
      }

while (1)
      {
      /*
       * The interesting bit.  IFFPARSE_RAWSTEP permits us to have precision monitoring of the
       * parsing process, which is necessary if we wish to print the structure of an IFF file.
       * ParseIFF() with _RAWSTEP will return the following things for the following reasons:
       *
       * Return code:              Reason:
       * 0                    Entered new context.
       * IFFERR_EOC                About to leave a context.
       * IFFERR_EOF                Encountered end-of-file.
       * <anything else>           A parsing error.
       */
      error = ParseIFF (iff, IFFPARSE_RAWSTEP);

      /*
       * Since we're only interested in when we enter a context, we "discard" end-of-context
       * (_EOC) events.
       */
      if (error == IFFERR_EOC)
            continue;
      else if (error)
            /*
             * Leave the loop if there is any other error.
             */
            break;
```

```
            /* If we get here, error was zero. Print out the current state of affairs. */
            PrintTopChunk (iff);
            }

        /*
         * If error was IFFERR_EOF, then the parser encountered the end of
         * the file without problems.  Otherwise, we print a diagnostic.
         */
        if (error == IFFERR_EOF)
            puts ("File scan complete.");
        else
            printf ("File scan aborted, error %ld: %s\n",
                error, errormsgs[-error - 1]);

bye:
        if (iff) {
            /* Terminate the IFF transaction with the stream.  Free all associated structures. */
            CloseIFF (iff);

            /*
             * Close the stream itself.
             */
            if (iff->iff_Stream)
                if (cbio)
                    CloseClipboard ((struct ClipboardHandle *)
                                iff->iff_Stream);
                else
                    Close (iff->iff_Stream);

            /* Free the IFF_File structure itself. */
            FreeIFF (iff);
            }
        if (IFFParseBase) CloseLibrary (IFFParseBase);

        exit (RETURN_OK);
}

void
PrintTopChunk (iff)
struct IFFHandle *iff;
{
        struct ContextNode      *top;
        short                   i;
        char                    idbuf[5];

        /* Get a pointer to the context node describing the current context. */
        if (!(top = CurrentChunk (iff)))
            return;

        /*
         * Print a series of dots equivalent to the current nesting depth of chunks processed so far.
         * This will cause nested chunks to be printed out indented.
         */
        for (i = iff->iff_Depth;  i--; )
            printf (". ");

        /* Print out the current chunk's ID and size. */
        printf ("%s %ld ", IDtoStr (top->cn_ID, idbuf), top->cn_Size);

        /* Print the current chunk's type, with a newline. */
        puts (IDtoStr (top->cn_Type, idbuf));
}
```

Function Reference

The following are brief descriptions of the IFFParse functions discussed in this chapter. IFFParse library functions are avialable in Release 2 of the Amiga OS and are backward compatible with older versions of the system. Further information about these and other IFFParse functions can be found in the 3rd edition of the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*, also from Addison-Wesley.

Table 33-2: IFFParse Library Functions

Function	Description
AllocIFF()	Creates an **IFFHandle** structure.
FreeIFF()	Frees the **IFFHandle** structure created with **AllocIFF()**.
OpenIFF()	Initialize an **IFFHandle** structure to read or write an IFF stream.
CloseIFF()	Closes an IFF context.
InitIFF()	Initialize an **IFFHandle** as a user-defined stream.
InitIFFasDOS()	Initialize an **IFFHandle** as an AmigaDOS stream.
InitIFFasClip()	Initialize an **IFFHandle** as a clipboard stream.
OpenClipboard()	Create a handle on a clipboard unit for **InitIFFasClip()**.
ParseIFF()	Parse an IFF file from an **IFFHandle** stream.
ReadChunkBytes()	Read bytes from the current chunk into a buffer.
ReadChunkRecords()	Read record elements from the current chunk into a buffer.
StopChunk()	Declare a chunk that should cause **ParseIFF()** to return.
CurrentChunk()	Get the context node for the current chunk.
PropChunk()	Specify a property chunk to store.
FindProp()	Search for a stored property in a given context.
CollectionChunk()	Declare a chunk type for collection.
FindCollection()	Get a pointer to the current list of collection items.
StopOnExit()	Declare a stop condition for exiting a chunk.
EntryHandler()	Add an entry handler to the **IFFHandle** context.
ExitHandler()	Add an exit handler to the **IFFHandle** context.
PushChunk()	Push a given context node onto the top of the context stack.
PopChunk()	Pop the top context node off of the context stack.
CurrentChunk()	Get the top context node for the current chunk.
ParentChunk()	Get the nesting context node for a given chunk.
AllocLocalItem()	Create a **LocalContextItem** (**LCI**) structure.
LocalItemData()	Returns a pointer to the user data of a **LocalContextItem** (**LCI**).
StoreLocalItem()	Insert a **LocalContextItem** (**LCI**).
StoreItemInContext()	Store a **LocalContextItem** in a given context node.
FindPropContext()	Find the property context for the current state.
FindLocalItem()	Return a **LocalContextItem** from the context stack.
FreeLocalItem()	Free a **LocalContextItem** (**LCI**) created with **AllocLocalItem()**.
SetLocalItemPurge()	Set purge vector for a local context item.

Chapter 34
KEYMAP LIBRARY

Amiga computers are sold internationally with a variety of local keyboards which match the standards of particular countries. All Amigas have keyboards which are physically similar, and keys which output the same low-level raw key code for any particular physical key. However, in different countries, the keycaps of the keys may contain different letters or symbols. Since the physical position of a key determines the raw key code that it generates, raw key codes are not internationally compatible. For instance, on the German keyboard, the Y and Z keys are swapped when compared to the USA keyboard. The second key on the fifth row will generate the same raw key code on all Amiga keyboards, but should be decoded as a Z on a US keyboard and as a Y on a German keyboard.

The Amiga uses the ECMA-94 Latin1 International 8-bit character set, and can map raw key codes to any desired ANSI character value, string, or escape sequence. This allows national keyboards to be supported by using *keymaps*. A keymap is a file which describes what character or string is tied to what key code. Generally, the user's startup-sequence will set a system default keymap that is correct for the user's keyboard. The console.device translates the raw key codes into the correct characters based on the installed keymap. This includes the translation of special *deadkey* sequential key combinations to produce accented international characters.

Programs which perform keyboard input using the console.device, CON:, RAW:, or Intuition VANILLAKEY, will receive the correct ASCII values for a user's keycaps, based on their keymap. But some applications may require custom keymaps, or may need to perform their own translation between raw key codes and ANSI characters. In this chapter, the term ANSI refers to standard 8-bit character definitions which include printable ASCII characters, special characters, and escape sequences.

Until V37, keymapping commands were only available in the console.device. Keymap.library is a new library in Release 2 (V37). It offers the some of the keymap commands of the console.device, enabling applications to inquire after the default keymap and map key codes to ANSI characters. It also provides the ability to map ANSI characters back into raw codes. Unlike the console.device however, it can not be used to select a keymap for only one application, i.e., one console window.

As a prelude to the following material, note that the Amiga keyboard transmits raw key information to the computer in the form of a key position and a transition. Raw key positions range from hexadecimal 00 to 7F. When a key is released, its raw key position, plus hexadecimal 80, is transmitted.

Keymap Functions

Table 34-1: Keymap Library Functions

AskKeyMapDefault()	Ask for a pointer to the current default keymap
MapANSI()	Encode an ANSI string into key codes
MapRawKey()	Decode a raw key input event to an ANSI string
SetKeyMapDefault()	Set the current default keymap for the system

Table 34-2: Console Device Keymap Commands

CD_ASKKEYMAP	Ask for the current console's keymap
CD_SETKEYMAP	Set the current console's keymap
CD_ASKDEFAULTKEYMAP*	Set the current default keymap
CD_SETDEFAULTKEYMAP**	Ask for a pointer to the current default keymap
* Obsolete - use AskKeyMapDefault() ** Obsolete - use SetKeyMapDefault()	

All of these commands deal with a set of pointers to key translation arrays, known as a **KeyMap** structure. The **KeyMap** structure is defined in *<devices/keymap.h>* and is shown below.

```
struct KeyMap
    {
    UBYTE *km_LoKeyMapTypes;
    ULONG *km_LoKeyMap;
    UBYTE *km_LoCapsable;
    UBYTE *km_LoRepeatable;
    UBYTE *km_HiKeyMapTypes;
    ULONG *km_HiKeyMap;
    UBYTE *km_HiCapsable;
    UBYTE *km_HiRepeatable;
    };
```

The KeyMap structure contains pointers to arrays which define the ANSI character or string that should be produced when a key or a combination of keys are pressed. For example, a keymap might specify that the key with raw value hex 20 should produce the ANSI character ''a'', and if the Shift key is being held it should produce the character ''A''.

ASKING FOR THE DEFAULT KEYMAP

The **AskKeyMapDefault**() returns a pointer to the current default keymap. To use the mapping functions in keymap.library it is normally not necessary to call this function. They accept NULL as equivalent to this pointer for example to cache the system default in order to temporarily change the keymap your applications uses, or find the keymap in the keymap.resource list of loaded keymaps. You should never change the system wide default keymap unless the user asks you to do so; since the Amiga is a multitasking system, changing the keymap could interfere with the behaviour of other applications.

SETTING THE DEFAULT KEYMAP

The system default keymap can be set with the **SetKeyMapDefault()** function. This function takes a pointer to a loaded keymap. In general, this function should never be used by an application unless the application is a system Preferences editor, or an application that takes over the system. Normal applications should instead attach a console.device unit to their own Intuition window (see the *Devices* volume), and use the console.device command **CD_SETKEYMAP** to set a keymap only for their own console.

When making a keymap the system default, first check whether the keymap has been loaded previously by checking the keymap list of the keymap.resource. If it has not been loaded already, it can be loaded from devs:Keymaps, and added to the keymap list of keymap.resource. This will ensure that other applications which may want the keymap will not have to load a second instance. Once made the default, the keymap can never be safely removed from memory, even after if it is no longer the default, since other applications may still have and use a pointer to it.

ACCESSING THE KEYMAP FOR THE CURRENT CONSOLE

The function **AskKeyMap()** shown below does not return a pointer to a table of pointers to currently assigned key mapping. Instead, it *copies* the current set of pointers to a user-designated area of memory. **AskKeyMap()** returns a TRUE or FALSE value that says whether or not the function succeeded.

The function **SetKeyMap()**, also shown below, copies the designated key map data structure to the console device. Thus this routine is complementary to **AskKeymap()** in that it can restore an original key mapping as well as establish a new one.

> *Ask/SetKeyMap() functions.* These functions assume that you have already opened the *console.device* and that *request* is a valid IOStdReq structure for the newly opened console. These functions are not part of the keymap.library, nor of the console.device. These merely demonstrate **CD_ASKKEYMAP** and **CD_SETKEYMAP** which are console.device commands.

```
/* These functions require that you have created a port and an IO request,
 * and have opened the console device as shown in the Console Device chapter
 * of the Devices volume of this manual set.
 */
#include <devices/keymap.h>

BOOL AskKeyMap(struct IOStdReq *request, struct KeyMap *keymap)
{
    request->io_Command = CD_ASKKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap;  /* where to put it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else  return(TRUE); /* if no error, it worked. */
}

BOOL SetKeyMap(struct IOStdReq *request,struct KeyMap *keymap)
{
    request->io_Command = CD_SETKEYMAP;
    request->io_Length = sizeof(struct KeyMap);
    request->io_Data = (APTR)keymap;      /* where to get it */
    DoIO(request);
    if(request->io_Error) return(FALSE);
    else  return(TRUE);     /* if no error, it worked. */
}
```

MAPPING KEY CODES TO ANSI STRINGS

MapRawKey() is converts raw key codes to ANSI characters based on a default or supplied keymap.

```
WORD MapRawKey(struct InputEvent *inputevent, UBYTE *buffer,
               WORD bufferlength, struct Keymap *keymap);
```

MapRawKey() takes an IECLASS_RAWKEY inputevent, which may be chained, and converts the key codes to ANSI characters which are placed in the specified buffer. If the buffer would overflow, for example because a longer string is attached to a key, -1 will be returned. If no error occurred, **MapRawKey()** will return the number of bytes written in the buffer. The keymap argument can be set to NULL if the default keymap is to be used for translation, or can be a pointer to a specific **KeyMap** structure.

The following example shows how to implement the **MapRawKey()** function.

```
;/* maprawkey.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 maprawkey.c
Blink FROM LIB:c.o,maprawkey.o TO maprawkey LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
* maprawkey.c - Map Intuition RAWKEY events to ANSI with MapRawKey();
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <intuition/intuition.h>
#include <devices/inputevent.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/keymap_protos.h>
#include <clib/intuition_protos.h>

#include <stdio.h>
#include <stdlib.h>

#ifdef LATTICE
int CXBRK(void)   { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }   /* really */
#endif

/* our function prototypes */
void openall(void);
void closeall(void);
void closeout(UBYTE *errstring, LONG rc);

struct Library    *IntuitionBase = NULL;
struct Library    *KeymapBase    = NULL;
struct Window     *window        = NULL;

void main(int argc, char **argv)
{
    struct IntuiMessage *imsg;
    APTR                *eventptr;
    struct InputEvent   inputevent = {0};
    LONG                windowsignal;
    UBYTE               buffer[8];
    COUNT               i;
    BOOL                Done = FALSE;

    openall();

    window = OpenWindowTags(NULL,
                            WA_Width,  500,
                            WA_Height, 60,
                            WA_Title, "MapRawKey - Press Keys",
                            WA_Flags, WFLG_CLOSEGADGET | WFLG_ACTIVATE,
                            WA_IDCMP, IDCMP_RAWKEY | IDCMP_CLOSEWINDOW,
                            TAG_DONE);
    if(window == NULL)   closeout("Can't open window",RETURN_FAIL);
```

```
    windowsignal = 1L << window->UserPort->mp_SigBit;

    /* Initialize InputEvent structure (already cleared to 0) */
    inputevent.ie_Class = IECLASS_RAWKEY;

    while(!Done)
       {
       Wait(windowsignal);

       while (imsg = (struct IntuiMessage *)GetMsg(window->UserPort))
           {
           switch(imsg->Class)
               {
                case IDCMP_CLOSEWINDOW:
                    Done = TRUE;
                    break;

                case IDCMP_RAWKEY:
                    inputevent.ie_Code = imsg->Code;
                    inputevent.ie_Qualifier = imsg->Qualifier;

                    printf("RAWKEY: Code=$%04x  Qualifier=$%04lx\n",
                             imsg->Code, imsg->Qualifier);

                    /* Make sure deadkeys and qualifiers are taken
                     * into account.
                     */
                    eventptr = imsg->IAddress;
                    inputevent.ie_EventAddress = *eventptr;

                    /* Map RAWKEY to ANSI */
                    i = MapRawKey(&inputevent, buffer, 8, NULL);

                    if (i == -1) Write(Output(),"*Overflow*",10);
                    else if (i)
                        {
                        /* This key or key combination mapped to something */
                        printf("MAPS TO: ");
                        Write(Output(),buffer,i);
                        printf("\n");
                        }
                    break;
               }
            ReplyMsg((struct Message *)imsg);
            }
        }
    CloseWindow(window);
    closeall();
    exit(RETURN_OK);
}

void openall(void)
{

    KeymapBase = OpenLibrary("keymap.library", 37);
    if (KeymapBase == NULL)    closeout("Kickstart 2.0 required",RETURN_FAIL);

    IntuitionBase = OpenLibrary("intuition.library", 37);
    if (IntuitionBase == NULL) closeout("Can't open intuition",RETURN_FAIL);
}

void closeall(void)
{
    if (IntuitionBase)  CloseLibrary(IntuitionBase);
    if (KeymapBase)     CloseLibrary(KeymapBase);
}

void closeout(UBYTE *errstring, LONG rc)
{
    if (*errstring)  printf("%s\n",errstring);
    closeall();
    exit(rc);
}
```

MAPPING ANSI STRINGS TO KEY CODES

The **MapANSI()** function translates ANSI strings into raw key codes, complete with qualifiers and (double) dead keys, based on a default or supplied keymap.

```
LONG MapANSI(UBYTE *string, LONG stringlength, UBYTE *buffer,
             LONG bufferlength, struct KeyMap *keymap);
```

The string argument is a pointer to an ANSI string, of length stringlength. The buffer argument is a pointer to the memory block where the translated key codes will be placed. The length of this buffer must be indicated in WORDs since each translation will occupy one byte for the key code and one for the qualifier. Since one ANSI character can be translated to two dead keys and one key, the buffer must be at least 3 WORDs per character in the string to be translated. The keymap argument can be set to NULL if the default keymap is to be used, or can be a pointer to a **KeyMap** structure. Upon return, the function will indicate how many key code/qualifier combinations are placed in the buffer or a negative number in case an error occurred. If zero is returned, the character could not be translated.

The following example shows the usage of **MapANSI()** and demonstrates how returned key codes can be processed.

```
;/* mapansi.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfistq -v -y -j73 mapansi.c
Blink FROM LIB:c.o,mapansi.o TO mapansi LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

mapansi.c - converts a string to input events using MapANSI() function.

            This example will also take the created input events
            and add them to the input stream using the simple
            commodities.library function AddIEvents().  Alternately,
            you could open the input.device and use the input device
            command IND_WRITEEVENT to add events to the input stream.
*/

#include      <exec/types.h>
#include      <exec/memory.h>
#include      <exec/io.h>
#include      <dos/dos.h>
#include      <devices/input.h>
#include      <devices/inputevent.h>

#include      <clib/exec_protos.h>
#include      <clib/dos_protos.h>
#include      <clib/keymap_protos.h>
#include      <clib/commodities_protos.h>

#include      <stdio.h>
#include      <stdlib.h>

#ifdef LATTICE
int CXBRK(void)   { return(0); }  /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }   /* really */
#endif

struct Library          *KeymapBase = NULL;       /* MapAnsi() function    */
struct Library          *CxBase = NULL;           /* AddIEvents() function */

struct InputEvent       *InputEvent = NULL;       /* we'll allocate this */

/* prototypes for our program functions */

void openall(void);
void closeall(void);
void closeout(UBYTE *errstring, LONG rc);
```

```
void main(int argc, char **argv)
{
    UBYTE               *string;
    UBYTE               *tmp1;
    UBYTE               *tmp2;
    UBYTE               iebuffer[6];           /* Space for two dead keys */
                                               /* + 1 key + qualifiers    */
    COUNT               i;
    LONG                rc = 0;


    openall();

    string = ";String converted to input events and sent to input device\n";

    InputEvent->ie_Class = IECLASS_RAWKEY;

    /* Turn each character into an InputEvent */
    tmp1 = string;

    while (*tmp1)
    {
        /* Convert one character, use default key map */
        i = MapANSI(tmp1, 1, iebuffer, 3, NULL);

        /* Make sure we start without deadkeys */
        InputEvent->ie_Prev1DownCode = InputEvent->ie_Prev1DownQual = 0;
        InputEvent->ie_Prev2DownCode = InputEvent->ie_Prev2DownQual = 0;

        tmp2 = iebuffer;

        switch (i)
        {
          case -2:
            printf("Internal error\n", NULL);
            rc = RETURN_FAIL;
            break;

          case -1:
            printf("Overflow\n", NULL);
            rc = RETURN_FAIL;
            break;

          case 0:
            printf("Can't generate code\n", NULL);
            break;

          case 3:
            InputEvent->ie_Prev2DownCode = *tmp2++;
            InputEvent->ie_Prev2DownQual = *tmp2++;
            /* FALL THROUGH */

          case 2:
            InputEvent->ie_Prev1DownCode = *tmp2++;
            InputEvent->ie_Prev1DownQual = *tmp2++;
            /* FALL THROUGH */

          case 1:
            InputEvent->ie_Code = *tmp2++;
            InputEvent->ie_Qualifier = *tmp2;
            break;
        }

        if (rc == RETURN_OK)
        {
            /* Send the key down event */
            AddIEvents(InputEvent);

            /* Create a key up event */
            InputEvent->ie_Code |= IECODE_UP_PREFIX;

            /* Send the key up event */
            AddIEvents(InputEvent);
         }

        if (rc != RETURN_OK)
```

```
            break;

        tmpl++;
    }

    closeall();
    exit(rc);
}


void openall(void)
{
    KeymapBase = OpenLibrary("keymap.library", 37);
    if (KeymapBase == NULL)  closeout("Kickstart 2.0 required", RETURN_FAIL);

    CxBase = OpenLibrary("commodities.library", 37);
    if (CxBase == NULL)  closeout("Kickstart 2.0 required", RETURN_FAIL);

    InputEvent = AllocMem(sizeof(struct InputEvent), MEMF_CLEAR);
    if (InputEvent == NULL)  closeout("Can't allocate input event",RETURN_FAIL);
}


void closeall()
{
    if (InputEvent)    FreeMem(InputEvent, sizeof(struct InputEvent));
    if (CxBase)        CloseLibrary(CxBase);
    if (KeymapBase)    CloseLibrary(KeymapBase);
}


void closeout(UBYTE *errstring, LONG rc)
{
    if(*errstring)     printf("%s\n",errstring);
    closeall();
    exit(rc);
}
```

DETAILS OF THE KEYMAP STRUCTURE

A **KeyMap** structure contains pointers to arrays which determine the translation from raw key codes to ANSI characters.

```
struct KeyMap
    {
    UBYTE *km_LoKeyMapTypes;
    ULONG *km_LoKeyMap;
    UBYTE *km_LoCapsable;
    UBYTE *km_LoRepeatable;
    UBYTE *km_HiKeyMapTypes;
    ULONG *km_HiKeyMap;
    UBYTE *km_HiCapsable;
    UBYTE *km_HiRepeatable;
    };
```

LoKeyMap and HighKeyMap

The low key map provides translation of the key values from hex 00-3F; the high key map provides translation of key values from hex 40-7F. Key values from hex 68-7F are not used by the existing keyboards, but this may change in the future. A raw key value (hex 00-7F) plus hex 80 is the release of that key. If you need to check for raw key releases do it like this:

```
if (keyvalue & 0x80)     {  /* do key up processing   */  }
else                     {  /* do key down processing */  }
```

Raw output from the keyboard for the low key map does not include the space bar, Tab, Alt, Ctrl, arrow keys, and several other keys.

Table 34-3: High Key Map Hex Values

Key Number	Keycap Legend or Function	Key Number	Keycap Legend or Function
40	Space	50-59	Function keys F1-F10
41	Backspace	5A-5E	Numeric Pad characters
42	Tab	5F	Help
43	Enter	60	Left Shift
44	Return	61	Right Shift
45	Escape	62	Caps Lock
46	Delete	63	Control
4A	Numeric Pad character	64	Left Alt
4C	Cursor Up	65	Right Alt
4D	Cursor Down	66	Left Amiga
4E	Cursor Right	67	Right Amiga
4F	Cursor Left		

The keymap table for the low and high keymaps consists of 4-byte entries, one per hex key code. These entries are interpreted in one of two possible ways:

- As four separate bytes, specifying how the key is to be interpreted when pressed alone, with one qualifier, with another qualifier, or with both qualifiers (where a qualifier is one of three possible keys: Ctrl, Alt, or Shift).

- As a longword containing the address of a string descriptor, where a string of characters is to be output when this key is pressed. If a string is to be output, any combination of qualifiers can affect the string that may be transmitted.

- As a longword containing the address of a dead-key descriptor, where additional data describe the character to be output when this key is pressed alone or with another dead key.

The keymap tables must be word aligned. The keymap tables *must* begin aligned on a word boundary. Each entry is four bytes long, thereby maintaining word alignment throughout the table. This is necessary because some of the entries may be longword addresses and *must* be aligned properly for the 68000.

LoKeyMapTypes and HiKeyMapTypes

The tables named **km_LoKeyMapTypes** and **km_HiKeyMapTypes** each contain one byte per raw key code. Each byte defines the type of entry that is found in the keymap table for that raw key code.

Possible key types are:

- Any of the qualifier groupings noted above

- KCF_STRING + any combination of KCF_SHIFT, KCF_ALT, KCF_CONTROL (or KC_NOQUAL) if the result of pressing the key is to be a stream of bytes (and key-with-one-or-more-qualifiers is to be one or more alternate streams of bytes).

  Any key can be made to output up to eight unique byte streams if KCF_STRING is set in its keytype. The only limitation is that the total length of all of the strings assigned to a key must be within the "jump range" of a single byte increment. See the "String-Output Keys" section below for more information.

- KCF_DEAD + any combination of KCF_SHIFT, KCF_ALT, KCF_CONTROL (or KC_NOQUAL) if the key is a dead-class key and can thus modify or be modified by another dead-class key. See the "Dead-Class Keys" section below for more information.

The low keytype table covers the raw key codes from hex 00-3F and contains one byte per key code. Therefore this table contains 64 (decimal) bytes. The high keytype table covers the raw key codes from hex 40-7F and contains 64 (decimal) bytes.

More About Qualifiers

For keys such as the Return key or Esc key, the qualifiers specified in the keytypes table (up to two) are the qualifiers used to establish the response to the key. This is done as follows. In the keytypes table, the values listed for the key types are those listed for the qualifiers in *<devices/keymap.h>* and *<devices/keymap.i>*. Specifically, these qualifier equates are:

KC_NOQUAL	0x00
KCF_SHIFT	0x01
KCF_ALT	0x02
KCF_CONTROL	0x04
KC_VANILLA	0x07
KCF_DOWNUP	0x08
KCF_STRING	0x40

As shown above, the qualifiers for the various types of keys occupy specific bit positions in the key types control byte. As you may have noticed, there are three possible qualifiers, but only a 4-byte space in the table for each key. This does not allow space to describe what the computer should output for all possible combinations of qualifiers. A solution exists, however, for "vanilla" keys, such as the alphabetic keys. Here is how that works.

Keys of type KC_VANILLA use the 4 bytes to represent the data output for the key alone, Shifted key, Alt'ed key, and Shifted-and-Alt'ed key. Then for the Ctrl-key-plus-vanilla-key, use the code for the key alone with bits 6 and 5 set to 0.

> *The Vanilla Qualifier Does Not Mean Plain.* The qualifier KC_VANILLA is equivalent to KCF_SHIFT+KCF_ALT+KCF_CONTROL.

This table shows how to interpret the keymap for various combinations of the qualifier bits:

Table 34-4: Keymap Qualifier Bits

If Keytype is:	Then data at this position in the keytable is output when the key is pressed along with:			
KC_NOQUAL	-	-	-	alone
KCF_SHIFT	-	-	Shift	alone
KCF_ALT	-	-	Alt	alone
KCF_CONTROL	-	-	Ctrl	alone
KCF_ALT+KCF_SHIFT	Shift+Alt	Alt	Shift	alone
KCF_CONTROL+KCF_ALT	Ctrl+Alt	Ctrl	Alt	alone
KCF_CONTROL+KCF_SHIFT	Ctrl+Shift	Ctrl	Shift	alone
KC_VANILLA	Shift+Alt	Alt	Shift	alone*

*Special case—Ctrl key, when pressed with one of the alphabet keys and certain others, is to output key-alone value with the bits 6 and 5 set to zero.

String Output Keys

When a key is to output a string, the keymap table contains the address of a string descriptor in place of a 4-byte mapping of a key as shown above. Here is a partial table for a new high keymap table that contains only three entries thus far. The first two are for the space bar and the backspace key; the third is for the tab key, which is to output a string that says "[TAB]". An alternate string, "[SHIFTED-TAB]", is also to be output when a shifted TAB key is pressed.

```
newHiMapTypes:
    DC.B        KCF_ALT,KC_NOQUAL,      ;key 41
    DC.B        KCF_STRING+KCF_SHIFT,   ;key 42
      ...       ; (more)
newHiMap:
    DC.B        0,0,$A0,$20     ;key 40: space bar, and Alt-space bar
    DC.B        0,0,0,$08       ;key 41: Back Space key only
    DC.L        newkey42        ;key 42: new definition for string to output for Tab key
      ...       ; (more)
newkey42:
    DC.B        new42ue - new42us       ;length of the unshifted string
    DC.B        new42us - newkey42      ;number of bytes from start of
                                        ;string descriptor to start of this string
    DC.B        new42se - new42ss       ;length of the shifted string
    DC.B        new42ss - newkey42      ;number of bytes from start of
                                        ;string descriptor to start of this string
new42us:  DC.B        '[TAB]'
new42ue:
new42ss:  DC.B        '[SHIFTED-TAB]'
new42se:
```

The new high map table points to the string descriptor at address newkey42. The new high map types table says that there is one qualifier, which means that there are two strings in the key string descriptor.

Each string in the descriptor takes two bytes in this part of the table: the first byte is the length of the string, and the second byte is the distance from the start of the descriptor to the start of the string. Therefore, a single string (KCF_STRING + KC_NOQUAL) takes 2 bytes of string descriptor. If there is one qualifier, 4 bytes of descriptor are used. If there are two qualifiers, 8 bytes of descriptor are used. If there are 3 qualifiers, 16 bytes of descriptor are used. All strings start immediately following the string descriptor in that they are accessed as single-byte offsets from the start of the descriptor itself. Therefore, the distance from the start of the descriptor to the last string in the set (the one that uses the entire set of specified qualifiers) must start within 255 bytes of the descriptor address.

Because the length of the string is contained in a single byte, the length of any single string must be 255 bytes or less while also meeting the "reach" requirement. However, the console input buffer size limits the string output from any individual key to 32 bytes maximum.

The length of a keymap containing string descriptors and strings is variable and depends on the number and size of the strings that you provide.

Capsable Bit Tables

The vectors **km_LoCapsable** and **km_HiCapsable** each point to an array of 8 bytes that contain more information about the keytable entries. Specifically, if the Caps Lock key has been pressed (the Caps Lock LED is on) and if there is a bit *on* in that position in the capsable map, then this key will be treated as though the Shift key is now currently pressed. For example, in the default key mapping, the alphabetic keys are "capsable" but the punctuation keys are not. This allows you to set the Caps Lock key, just as on a normal typewriter, and get all capital letters. However, unlike a normal typewriter, you need not go out of Caps Lock to correctly type the punctuation symbols or numeric keys.

In the byte array, the bits that control this feature are numbered from the lowest bit in the byte, and from the lowest memory byte address to the highest. For example, the bit representing capsable status for the key that transmits raw code 00 is bit 0 in byte 0; for the key that transmits raw code 08 it is bit 0 in byte 1, and so on.

There are 64 bits (8 bytes) in each of the two capsable tables.

Repeatable Bit Tables

The vectors **km_LoRepeatable** and **km_HiRepeatable** each point to an array of 8 bytes that contain additional information about the keytable entries. A bit for each key indicates whether or not the specified key should repeat at the rate set by the Input Preferences program.

The bit positions correspond to those specified in the capsable bit table. If there is a 1 in a specific position, the key can repeat. There are 64 bits (8 bytes) in each of the two repeatable tables.

KEY MAP STANDARDS

Users and programs depend on certain predictable behaviors from all keyboards and keymaps. With the exception of dead-class keys (see "Dead-Class Keys" section), mapping of keys in the low key map should follow these general rules:

- When pressed alone, keys should transmit the ASCII equivalent of the unshifted letter or lower symbol on the keycap.
- When Shifted, keys should transmit the ASCII equivalent of the shifted letter or upper symbol printed on the keycap.
- When Alt'ed, keys should generally transmit the same character (or act as the same deadkey) as the Alt'ed key in the usa1 keymap.
- When pressed with CTRL alone, alphabetic keys should generally transmit their unshifted value but with bits 5 and 6 cleared. This allows keyboard typing of "control characters." For example, the C key (normally value $63) should transmit value $03 (Ctrl-C) when Ctrl and C are pressed together.

The keys in the high key map (keys with raw key values $40 and higher) are generally non-alphanumeric keys such as those used for editing (backspace, delete, cursor keys, etc.), and special Amiga keys such as the function and help keys. Keymaps should translate these keys to the same values or strings as those shown in table 34-6, ROM Default Key Mapping.

In addition to their normal unshifted and shifted values, the following translations are standard for particular qualified high keymap keys:

Key	Generates This Value	If Used with Qualifier, Generates This Value
Space	$20	$A0 with qualifier KCF_ALT
Return	$0D	$0A with qualifier KCF_CONTROL
Esc	$1B	$9B with qualifier KCF_ALT

DEAD-CLASS KEYS

All of the national keymaps, including USA, contain *dead-class* keys. This term refers to keys that either modify or can themselves be modified by other dead-class keys. There are two types of dead-class keys: dead and deadable. A dead key is one which can modify certain keys pressed immediately following. For example, on the German keyboard there is a dead key marked with the grave accent (`). The dead key produces no console output, but when followed by (for instance) the A key, the combination will produce the a-grave (à) character (National Character Code $E0). On the U.S. keyboard, Alt-G is the deadkey used to add the grave accent (`) to the next appropriate character typed. A deadable key is one that can be prefixed by a dead key. The A key in the previous example is a deadable key. Thus, a dead key can only affect the output of a deadable key.

For any key that is to have a dead-class function, whether dead or deadable, the qualifier KCF_DEAD flag must be included in the entry for the key in the KeyMapTypes table. The KCF_DEAD type may also be used in conjunction with the other qualifiers. Furthermore, the key's keymap table entry must contain the longword address of the key's dead-key descriptor data area in place of the usual 4 ASCII character mapping.

Below is an excerpt from the Amiga 1000 German key map which is referred to in the following discussion.

```
KMLowMapType:
        DC.B    KCF_DEAD+KC_VANILLA     ; aA (Key 20)
                    ...                 ; (more...)
        DC.B    KCF_DEAD+KC_VANILLA     ; hH (Key 25)
                    ...                 ; (more...)
KMLowMap:
        DC.L    key20                   ; a, A, ae, AE
                    ...                 ; (more...)
        DC.L    key25                   ; h, H, dead ^
                    ...                 ; (more...)
;------ possible dead keys
key25:
        DC.B    0,'h',0,'H'             ; h, H
        DC.B    DPF_DEAD,3,DPF_DEAD,3   ; dead ^, dead ^
        DC.B    0,$08,0,$08,0,$88,0,$88 ; control translation
                    ...                 ; (more...)
;------ deadable keys (modified by dead keys)
key20:                  ; a, A, ae, AE
        DC.B    DPF_MOD,key20u-key20    ; deadable flag, number of
                                        ; bytes from start of key20
                                        ; descriptor to start of un-
                                        ; shifted data
        DC.B    DPF_MOD,key20s-key20    ; deadable flag, number of
                                        ; bytes from start of key20
                                        ; descriptor to start of shift-
                                        ; ed data
        DC.B    0,$E6,0,$C6             ; null flags followed by rest
        DC.B    0,$01,0,$01,0,$81,0,$81 ; of values (ALT, CTRL...)
key20u:
        DC.B    'a',$E1,$E0,$E2,$E3,$E4 ; 'a' alone and characters to
                                        ; output when key alone is
                                        ; prefixed by a dead key
        DC.B    $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
        DC.B    $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is `
key20s:
        DC.B    'A',$C1,$C0,$C2,$C3,$C4 ; SHIFTed 'a' and characters to
                                        ; output when SHIFTed key is
                                        ; prefixed by a dead key
        DC.B    $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
        DC.B    $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is `
```

In the example, key 25 (the H key) is a dead key and key 20 (the A key) is a deadable key. Both keys use the addresses of their descriptor data areas as entries in the LoKeyMap table. The LoKeyMapTypes table says that there are four qualifiers for both: the requisite KCF_DEAD, as well as KCF_SHIFT, KCF_ALT, and KCF_CONTROL. The number of qualifiers determine length and arrangement of the descriptor data areas for each key. The next table shows how to interpret the KeyMapTypes for various combinations of the qualifier bits. For each possible position a pair of bytes is needed. The first byte in each pair tells how to interpret the second byte (more about this below).

Table 34-5: Dead Key Qualifier Bits

If type is:	Then the pair of bytes in this position in the dead-class key descriptor data is output when the key is pressed along with:							
NOQUAL	alone	-	-	-	-	-	-	-
A	alone	A	-	-	-	-	-	-
C	alone	C	-	-	-	-	-	-
S	alone	S	-	-	-	-	-	-
A+C	alone	A	C	A+C	-	-	-	-
A+S	alone	S	A	A+S	-	-	-	-
C+S	alone	S	C	C+S	-	-	-	-
S+A+C (VANILLA)	alone	S	A	S+A	C	C+S	C+A	C+S+A
The abbreviations A, C, S stand for KCF_ALT, KCF_CONTROL, and KCF_SHIFT, respectively. Also note that the ordering is reversed from that in the normal **KeyMap** table.								

Because keys 20 and 25 each use three qualifier bits (not including KCF_DEAD), according to the table there must be 8 pairs of data, arranged as shown. Had only KCF_ALT been set, for instance, (not including KCF_DEAD), just two pairs would have been needed.

As mentioned earlier, the first byte of each data pair in the descriptor data area specifies how to interpret the second byte. There are three possible values: 0, DPF_DEAD and DPF_MOD. In the Amiga 1000 German keymap listed above, DPF_DEAD appears in the data for key 25, while DPF_MOD is used for key 20. It is the use of these flags that determines whether a dead-class key has dead or deadable function. A value of zero causes the unrestricted output of the following byte.

If the flag byte is DPF_DEAD, then that particular key combination (determined by the placement of the pair of bytes in the data table) is dead and will modify the output of the next key pressed (if deadable). How it modifies is controlled by the second byte of the pair which is used as an index into part(s) of the data area for ALL the deadable (DPF_MOD set) keys.

Before going further, an understanding of the structure of a descriptor data area wherein DPF_MOD is set for one (or more) of its members is necessary. Referring to the example, we see that DPF_MOD is set for the first and second pairs of bytes. According to its LoKeyMapTypes entry, and using table 34-5 (Dead Key Qualifier Bits) as a guide, these pairs represent the alone and SHIFTed values for the key. When DPF_MOD is set, the byte immediately following the flag must be the offset from the start of the key's descriptor data area to the start of a table of bytes describing the characters to output when this key combination is preceded by any dead keys. This is where the index mentioned above comes in. The value of the index from a prefixing dead key is used to determine which of the bytes from the deadable keys special table to output. The byte in the index+1 position is sent out. (The very first byte is the value to output if the key was not prefixed by a dead key.) Thus, if Alt-H is pressed (dead) and then Shift-A, an 'a' with a circumflex (^) accent will be output. This is because:

- The byte pair for the ALT position of the H key (key 25) is DPF_DEAD,3 so the index is 3.
- The byte pair for the SHIFT position of the A key (key 20) is DPF_MOD, key20s-key20, so we refer to the table-of-bytes at key20s.

- The third+1 byte of the table-of-bytes is $C2, an 'a' character.

A Note About Table Size. The number of bytes in the table-of-bytes for all deadable keys must be equal to the highest index value of all dead keys plus 1.

DOUBLE-DEAD KEYS

Double-dead keys are an extension of the dead-class keys explained above. Unlike normal dead keys wherein one dead key of type DPF_DEAD can modify a second of type DPF_MOD, double-dead keys employ two consecutive keys of type DPF_DEAD to together modify a third of type DPF_MOD.

For example, the key on the German keyboard labeled with single quotes ('') is a double-dead key. When this key is pressed alone and then pressed again shifted, there is no output. But when followed by an appropriate third key, for example the A key, the three keypresses combine to produce an 'a' with a circumflex (^) accent (character code $E2). Thus the double-dead pair qualify the output of the A key.

The system always keeps the last two down key codes for possible further translation. If they are both of type DPF_DEAD and the key immediately following is DPF_MOD then the two are used to form an index into the (third) key's translation table as follows:

In addition to the index found after the DPF_DEAD qualifier in a normal dead key, a second factor is included in the high nibble of double-dead keys (it is shifted into place with DP_2DFACSHIFT). Its value equals the total number of dead key types + 1 in the keymap. This second index also serves as an identifying flag to the system that two dead keys can be significant.

When a key of type DPF_MOD is pressed, the system checks the two key codes which preceded the current one. If they were both DPF_DEAD then the most recent of the two is checked for the double-dead index/flag. If it is found then a new index is formed by multiplying the value in lower nibble with that in the upper. Then, the lower nibble of the least recent DPF_DEAD key is added in to form the final offset.

Finally, this last value is used as an index into the translation table of the current, DPF_MOD, key.

The translation table of all deadable (DPF_MOD) keys has [number of dead key types + 1] * [number of double dead key types + 1] entries, arranged in [number of double dead key types + 1] rows of [number of dead key types + 1] entries. This is because as indices are assigned for dead keys in the keymap, those that are double dead keys are assigned the lower numbers.

Following is a code fragment from the German (d) keymap source:

```
key0C:
        DC.B    DPF_DEAD,1+(6<<DP_2DFACSHIFT)    ; dead '
        DC.B    DPF_DEAD,2+(6<<DP_2DFACSHIFT)    ; dead `
        DC.B    0,'=',0,'+'                      ; =, +
key20:                  ; a, A, ae, AE
        DC.B    DPF_MOD,key20u-key20,DPF_MOD,key20s-key20
        DC.B    0,$E6,0,$C6
        DC.B    0,$01,0,$01,0,$81,0,$81 ; control translation
key20u:
        DC.B    'a',$E1,$E0,$E2,$E3,$E4
        DC.B    $E1,$E1,$E2,$E1,$E1,$E1 ; most recent is '
        DC.B    $E0,$E2,$E0,$E0,$E0,$E0 ; most recent is `
key20s:
        DC.B    'A',$C1,$C0,$C2,$C3,$C4
        DC.B    $C1,$C1,$C2,$C1,$C1,$C1 ; most recent is '
        DC.B    $C0,$C2,$C0,$C0,$C0,$C0 ; most recent is `
```

Raw key0C, the German single quotes ('') key, is a double dead key. Pressing this key alone, then again while the shift key is down will produce no output but will form a double-dead qualifier. The output of key20 (A), a deadable key, will consequently be modified, producing an ''a'' with a circumflex (^) accent. The mechanics are as follows:

- When key0C is pressed alone the DPF_DEAD of the first byte pair in the key's table indicates that the key as dead. The second byte is then held by the system.

- Next, when key0C is pressed again, this time with the Shift key down, the DPF_DEAD of the second byte pair (recall that the second pair is used because of the SHIFT qualifier) again indicates the key is a dead key. The second byte of this pair is also held by the system.

- Finally, when the A key is pressed the system recalls the latter of the two bytes it has saved. The upper nibble, $6, is multiplied by the lower nibble, $2. The result, $0C, is then added to the lower nibble of the earlier of the two saved bytes, $1. This new value, $0D, is used as an index into the (unshifted) translation table of key20. The character at position $0D is character $E2, an 'a' with a circumflex (^) accent.

Note About Double Dead Keys. If only one double-dead key is pressed before a deadable key then the output is the same as if the double-dead were a normal dead key. If shifted key0C is pressed on the German keyboard and then immediately followed by key20, the output produced is character $E0, ' à '. As before, the upper nibble is multiplied with the lower, resulting in $0C. But because there was no second dead-key, this product is used as the final index.

Keyboard Layout

The keys with key codes $2B and $30 in the following keyboard diagrams are keys which are present on some national Amiga keyboards.

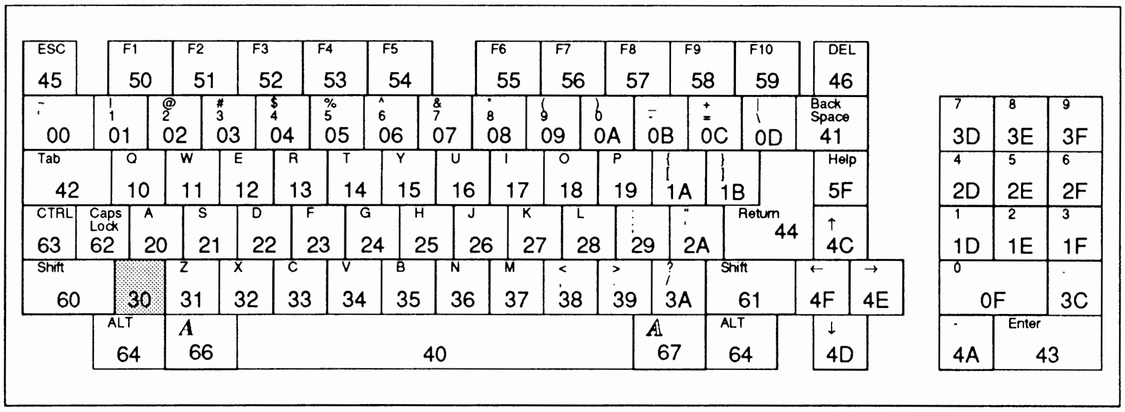


Figure 34-1: Amiga 1000 Keyboard Showing Key Codes in Hex

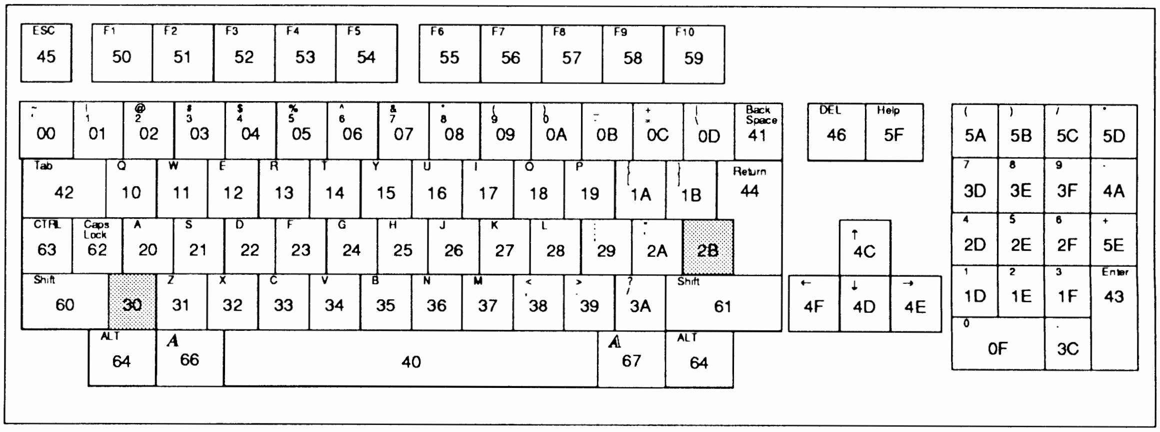


Figure 34-2: Amiga 500/2000/3000 Keyboard Showing Key Codes in Hex

The default values given above correspond to the values the console device will return when these keys are pressed with the keycaps as shipped with the standard American keyboard.

Table 34-6: ROM Default (USA0) and USA1 Console Key Mapping

Raw Key Number	Keycap Legend	Unshifted Default Value	Shifted Default Value
00	‘ ˜	‘ (Accent grave)	˜ (tilde)
01	1 !	1	!
02	2 @	2	@
03	3 #	3	#
04	4 $	4	$
05	5 %	5	%
06	6 ˆ	6	ˆ
07	7 &	7	&
08	8 *	8	*
09	9 (	9	(
0A	0)	0	)
0B	- _	- (Hyphen)	_ (Underscore)
0C	= +	=	+
0D	|		|
0E		(undefined)	
0F	0	0	0 (Numeric pad)
10	Q	q	Q
11	W	w	W
12	E	e	E
13	R	r	R
14	T	t	T
15	Y	y	Y
16	U	u	U
17	I	i	I
18	O	o	O
19	P	p	P
1A	[{	[	{
1B	] }	]	}
1C		(undefined)	
1D	1	1	1 (Numeric pad)
1E	2	2	2 (Numeric pad)
1F	3	3	3 (Numeric pad)
20	A	a	A
21	S	s	S
22	D	d	D
23	F	f	F
24	G	g	G
25	H	h	H
26	J	j	J
27	K	k	K
28	L	l	L
29	; :	;	:
2A	’ "	’ (single quote)	"
2B		(not on most US keyboards)	
2C		(undefined)	
2D	4	4	4 (Numeric pad)
2E	5	5	5 (Numeric pad)
2F	6	6	6 (Numeric pad)
30		(not on most US keyboards)	

Raw Key Number	Keycap Legend	Unshifted Default Value	Shifted Default Value
31	Z	z	Z
32	X	x	X
33	C	c	C
34	V	v	V
35	B	b	B
36	N	n	N
37	M	m	M
38	, <	, (comma)	<
39	. >	. (period)	>
3A	/ ?	/	?
3B		(undefined)	
3C	.	.	. (Numeric pad)
3D	7	7	7 (Numeric pad)
3E	8	8	8 (Numeric pad)
3F	9	9	9 (Numeric pad)
40	(Space bar)	20	20
41	Back Space	08	08
42	Tab	09	09
43	Enter	0D	0D (Numeric pad)
44	Return	0D	0D
45	Esc	1B	1B
46	Del	7F	7F
47		(undefined)	
48		(undefined)	
49		(undefined)	
4A	-	-	- (Numeric Pad)
4B		(undefined)	
4C	Up arrow	<CSI>A	<CSI>T
4D	Down arrow	<CSI>B	<CSI>S
4E	Forward arrow	<CSI>C	<CSI> A (note blank space after <CSI>)
4F	Backward arrow	<CSI>D	<CSI> @ (note blank space after <CSI>)
50	F1	<CSI>0~	<CSI>10~
51	F2	<CSI>1~	<CSI>11~
52	F3	<CSI>2~	<CSI>12~
53	F4	<CSI>3~	<CSI>13~
54	F5	<CSI>4~	<CSI>14~
55	F6	<CSI>5~	<CSI>15~
56	F7	<CSI>6~	<CSI>16~
57	F8	<CSI>7~	<CSI>17~
58	F9	<CSI>8~	<CSI>18~
59	F10	<CSI>9~	<CSI>19~
5A	(	(	((usa1 Numeric pad)
5B	)	)	) (usa1 Numeric pad)
5C	/	/	/ (usa1 Numeric pad)
5D	*	*	* (usa1 Numeric pad)
5E	+	+	+ (usa1 Numeric pad)
5F	HELP	<CSI>?~	<CSI>?~

Raw Key Number	Function or Keycap Legend	
60	Shift (left of space bar)	
61	Shift (right of space bar)	
62	Caps Lock	
63	Ctrl	
64	(Left) Alt	
65	(Right) Alt	
66	Amiga (left of space bar)	Left Amiga
67	Amiga (right of space bar)	Right Amiga
68	Left mouse button (not converted)	Inputs are only for the
69	Right mouse button (not converted)	mouse connected to Intuition,
6A	Middle mouse button (not converted)	(currently gameport one).
6B	(undefined)	
6C	(undefined)	
6D	(undefined)	
6E	(undefined)	
6F	(undefined)	
70-7F	(undefined)	
80-F8	Up transition (release or unpress key of one of the above keys) (80 for 00, F8 for 7F)	
F9	Last key code was bad (was sent in order to resynchronize)	
FA	Keyboard buffer overflow	
FB	(undefined, reserved for keyboard processor catastrophe)	
FC	Keyboard selftest failed	
FD	Power-up key stream start. Keys pressed or stuck at power-up will be sent between FD and FE.	
FE	Power-up key stream end	
FF	(undefined, reserved)	
FF	Mouse event, movement only, no button change (not converted)	

Notes about the preceding table:

1) ``<CSI>'' is the Control Sequence Introducer, value hex 9B.

2) ``(undefined)'' indicates that the current keyboard design should not generate this number. If you are using **SetKeyMap()** to change the key map, the entries for these numbers must still be included.

3) ``(not converted)'' refers to mouse button events. You must use the sequence ``<CSI>2{'' to inform the console driver that you wish to receive mouse events; otherwise these will not be transmitted.

4) ``(RESERVED)'' indicates that these key codes have been reserved for national keyboards. The $2B code key will be between the double-quote (") and Return keys. The $30 code key will be between the Shift and Z keys.

High Nybble \ Low Nybble	0000 00	0001 01	0010 02	0011 03	0100 04	0101 05	0110 06	0111 07	1000 08	1001 09	1010 0A	1011 0B	1100 0C	1101 0D	1110 0E	1111 0F
0000 00																
0001 10																
0010 20	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0011 30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100 40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101 50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0110 60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111 70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
1000 80																
1001 90																
1010 A0	NBSP	¡	¢	£	¤	¥	|	§	¨	©	ª	«	¬	SHY	®	¯
1011 B0	°	±	^{2}	^{3}	´	u	¶	•	¸	^{1}	º	»	1/4	1/2	3/4	¿
1100 C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
1101 D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	X	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
1110 E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
1111 F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure 34-3: ECMA-94 Latin1 International 8-Bit Character Set

Function Reference

The following chart gives a brief desription of the functions covered in this chapter. All of these functions require Release 2 or a later version of the Amiga operating system. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

Table 34-7: Keymap Library Functions

Function	Description
AskKeyMapDefault()	Get a pointer to the current system default keymap
MapANSI()	Convert ANSII string to raw key events
MapRawKey()	Convert raw key events to ANSII
SetKeyMapDefault()	Set the system default keymap

Chapter 35
MATH LIBRARIES

This chapter describes the structure and calling sequences required to access the Motorola Fast Floating Point (FFP), the IEEE single-precision math libraries and the IEEE double-precision math libraries via the Amiga-supplied interfaces.

In its present state, the FFP library consists of three separate entities: the basic math library, the transcendental math library, and C and assembly-language interfaces to the basic math library plus FFP conversion functions. The IEEE single-precision, introduced in Release 2, and the double-precision libraries each presently consists of two entities: the basic math library and the transcendental math library.

> *Open Each Library Separately.* Each **Task** using an IEEE math library must open the library itself. Library base pointers to these libraries may *not* be shared. Libraries can be context sensitive and may use the **Task** structure to keep track of the current context. Sharing of library bases by **Tasks** may seem to work in some systems. This is true for any of the IEEE math libraries.

Depending on the compiler used, it is not always necessary to explicitly call the library functions for basic floating point operations as adding, subtracting, dividing, etc. Consult the manual supplied with the compiler for information regarding the compiler options for floating point functions.

Math Libraries and Functions

There are six math libraries providing functions ranging from adding two floating point numbers to calculating a hyperbolic cosine. They are:

mathffp.library
: the basic function library

mathtrans.library
: the FFP transcendental math library

mathieeesingbas.library
: the IEEE single-precision library

mathieesingtrans.library
the IEEE single-precision transcendental library

mathieeedoubbas.library
the IEEE double-precision library

mathieesingtrans.library
the IEEE double-precision transcendental library

FFP Floating Point Data Format

FFP floating-point variables are defined within C by the float or FLOAT directive. In assembly language they are simply defined by a DC.L/DS.L statement. All FFP floating-point variables are defined as 32-bit entities (longwords) with the following format:

MMMMMMMM	MMMMMMMM	MMMMMMMM	EEEEEEE
31	23	15	7

The mantissa is considered to be a binary fixed-point fraction; except for 0, it is always normalized (the mantissa is shifted over and the exponent adjusted, so that the mantissa has a 1 bit in its highest position). Thus, it represents a value of less than 1 but greater than or equal to 1/2.

The sign bit is reset (0) for a positive value and set (1) for a negative value.

The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. It is held in excess-64 notation, which means that the two's-complement values are adjusted upward by 64, thus changing $40 (-64) through $3F (+63) to $00 through $7F. This facilitates comparisons among floating-point values.

The value of 0 is defined as all 32 bits being 0s. The sign, exponent, and mantissa are entirely cleared. Thus, 0s are always treated as positive.

The range allowed by this format is as follows:

DECIMAL:

$$9.22337177 * 10^{18} > +VALUE > 5.42101070 * 10^{-20}$$

$$-9.22337177 * 10^{18} < -VALUE < -2.71050535 * 10^{-20}$$

BINARY (HEXADECIMAL):

$$.FFFFFF * 2^{63} > +VALUE > .800000 * 2^{-63}$$

$$-.FFFFFF * 2^{63} < -VALUE < -.800000 * 2^{-64}$$

Remember that you cannot perform *any* arithmetic on these variables without using the fast floating-point libraries. The formats of the variables are *incompatible* with the arithmetic format of C-generated code; hence, all floating-point operations are performed through function calls.

FFP Basic Mathematics Library

The FFP basic math library contains entries for the basic mathematics functions such as add, subtract and divide. It resides in ROM and is opened by calling **OpenLibrary()** with "mathffp.library" as the argument.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>

struct Library *MathBase;

VOID main()
{
if (MathBase = OpenLibrary("mathffp.library", 0))
    {
          . . .

    CloseLibrary(MathBase);
    }
else
    printf("Can't open mathffp.library\n");
}
```

The global variable **MathBase** is used internally for all future library references.

FFP BASIC FUNCTIONS

SPAbs() `FLOAT SPAbs( FLOAT parm );`
Take absolute value of FFP variable.

SPAdd() `FLOAT SPAdd( FLOAT leftParm, FLOAT rightParm);`
Add two FFP variables.

SPCeil() `FLOAT SPCeil( FLOAT parm );`
Computer largest integer less than or equal to variable.

SPCmp() `LONG  SPCmp( FLOAT leftParm, FLOAT rightParm);`
Compare two FFP variables.

SPDiv() `FLOAT SPDiv( FLOAT leftParm, FLOAT rightParm);`
Divide two FFP variables.

SPFix() `LONG  SPFix( FLOAT parm );`
Convert FFP variable to integer.

SPFloor() `FLOAT SPFloor( FLOAT parm );`
Compute least integer greater than or equal to variable.

SPFlt() `FLOAT SPFlt( long integer );`
Convert integer variable to FFP.

SPMul() `FLOAT SPMul( FLOAT leftParm, FLOAT rightParm);`
Multiply two FFP variables.

SPNeg() `FLOAT SPNeg( FLOAT parm );`
Take two's complement of FFP variable.

SPSub() `FLOAT SPSub( FLOAT leftParm, FLOAT rightParm);`
Subtract two FFP variables.

SPTst() `LONG SPTst( FLOAT parm );`

Test an FFP variable against zero.

Be sure to include the proper data type definitions shown below.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>

struct Library *MathBase;

VOID main()
{
FLOAT f1, f2, f3;
LONG  i1;

if (MathBase = OpenLibrary("mathffp.library", 0))
    {
    i1 = SPFix(f1);             /* Call SPFix entry */
    f1 = SPFlt(i1);             /* Call SPFlt entry */

    if (SPCmp(f1,f2)) {};       /* Call SPCmp entry */
    if (!(SPTst(f1))) {};       /* Call SPTst entry */

    f1 = SPAbs(f2);             /* Call SPAbs entry */
    f1 = SPNeg(f2);             /* Call SPNeg entry */
    f1 = SPAdd(f2, f3);         /* Call SPAdd entry */
    f1 = SPSub(f2, f3);         /* Call SPSub entry */
    f1 = SPMul(f2, f3);         /* Call SPMul entry */
    f1 = SPDiv(f2, f3);         /* Call SPDiv entry */
    f1 = SPCeil(f2);            /* Call SPCeil entry */
    f1 = SPFloor(f2);           /* Call SPFloor entry */

    CloseLibrary(MathBase);
    }
else
    printf("Can't open mathffp.library\n");
}
```

The assembly language interface to the FFP basic math routines is shown below, including some details about how the system flags are affected by each operation. The access mechanism is:

```
MOVEA.L _MathBase,A6
JSR _LVOSPFix(A6)
```

FFP Basic Assembly Functions

Function	Input	Output	Condition Codes
_LVOSPAbs	D0 = FFP argument	D0 = FFP absolute value	N = 0 Z = 1 if result is zero V = 0 C = undefined X = undefined
_LVOSPAdd	D1 = FFP argument 1 D0 = FFP argument 2	D0 = FFP addition of arg1 + arg2	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

FFP Basic Assembly Functions

Function	Input	Output	Condition Codes
_LVOSPCeil	D0 = FFP argument	D0 = least integer >= argument	N = 1 if result is negative Z = 1 if result is zero V = undefined C = undefined Z = undefined
_LVOSPCmp	D1 = FFP argument 1 D0 = FFP argument 2	D0 = +1 if arg1 > arg2 D0 = -1 if arg1 < arg2 D0 = 0 if arg1 = arg2	N = 0 Z = 1 if result is zero V = 0 C = undefined X = undefined GT = arg2 > arg1 GE = arg2 >= arg1 EQ = arg2 = arg1 NE = arg2 <> arg1 LT = arg2 < arg1 LE = arg2 <= arg1
_LVOSPDiv	D1 = FFP argument 1 D0 = FFP argument 2	D0 = FFP division of arg2/arg1	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined
_LVOSPFix	D0 = FFP argument	D0 = Integer (two's complement)	N = 1 if result is negative Z = 1 if result is zero V = 1 if overflow occurred C = undefined X = undefined
_LVOSPFloor	D0 = FFP argument	D0 = largest integer <= argument	N = 1 if result is negative Z = 1 if result is zero V = undefined C = undefined Z = undefined
_LVOSPFlt	D0 = Integer (two's complement)	D0 = FFP result	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined
_LVOSPMul	D0 = FFP argument 1 D1 = FFP argument 2	D0 = FFP multiplication of arg1 * arg2	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined

FFP Basic Assembly Functions

Function	Input	Output	Condition Codes
_LVOSPNeg	D0 = FFP argument	D0 = FFP negated	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined
_LVOSPSub	D1 = FFP argument 1 D0 = FFP argument 2	D0 = FFP subtraction of arg2 - arg1	N = 1 if result is negative Z = 1 if result is zero V = 1 if result overflowed C = undefined Z = undefined
_LVOSPTst	D1 = FFP argument *Note*: This routine trashes the argument in D1.	D0 = +1 if arg > 0.0 D0 = -1 if arg < 0.0 D0 = 0 if arg = 0.0	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined EQ = arg = 0.0 NE = arg <> 0.0 PL = arg >= 0.0 MI = arg < 0.0

FFP Transcendental Mathematics Library

The FFP transcendental math library contains entries for the transcendental math functions sine, cosine, and square root. It resides on disk and is opened by calling **OpenLibrary()** with "mathtrans.library" as the argument.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>
#include <clib/mathtrans_protos.h>

struct Library *MathTransBase;

VOID main()
{
if (MathTransBase = OpenLibrary("mathtrans.library",0))
    {
          .
          .
          .
    CloseLibrary(MathTransBase);
    }
else
    printf("Can't open mathtrans.library\n");
}
```

The global variable **MathTransBase** is used internally for all future library references. Note that the transcendental math library is dependent upon the basic math library, which it will open if it is not open already. If you want to use the basic math functions in conjunction with the transcendental math functions however, you have to specifically open the basic math library yourself.

FFP TRANSCENDENTAL FUNCTIONS

SPAsin() `FLOAT SPAsin( FLOAT parm );`
Return arccosine of FFP variable.

SPAcos() `FLOAT SPAcos( FLOAT parm );`
Return arctangent of FFP variable.

SPAtan() `FLOAT SPAtan( FLOAT parm );`
Return arcsine of FFP variable.

SPSin() `FLOAT SPSin( FLOAT parm );`
Return sine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric sine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made.

SPCos() `FLOAT SPCos( FLOAT parm );`
Return cosine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric cosine value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made.

SPTan() `FLOAT SPTan( FLOAT parm );`
Return tangent of FFP variable. This function accepts an FFP radian argument and returns the trigonometric tangent value. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made.

SPSincos() `FLOAT SPSincos( FLOAT *cosResult, FLOAT parm);`
Return sine and cosine of FFP variable. This function accepts an FFP radian argument and returns the trigonometric sine as its result and the trigonometric cosine in the first parameter. If both the sine and cosine are required for a single radian value, this function will result in almost twice the execution speed of calling the **SPSin()** and **SPCos()** functions independently. For extremely large arguments where little or no precision would result, the computation is aborted and the "V" condition code is set. A direct return to the caller is made.

SPSinh() `FLOAT SPSinh( FLOAT parm );`
Return hyperbolic sine of FFP variable.

SPCosh() `FLOAT SPCosh( FLOAT parm );`
Return hyperbolic cosine of FFP variable.

SPTanh() `FLOAT SPTanh( FLOAT parm );`
Return hyperbolic tangent of FFP variable.

SPExp() `FLOAT SPExp( FLOAT parm );`
Return *e* to the FFP variable power. This function accepts an FFP argument and returns the result representing the value of *e* (2.71828...) raised to that power.

SPLog() `FLOAT SPLog( FLOAT parm );`
Return natural log (base *e*) of FFP variable.

SPLog10() `FLOAT SPLog10( FLOAT parm );`
Return log (base 10) of FFP variable.

SPPow() `FLOAT SPPow( FLOAT power, FLOAT arg );`
Return FFP arg2 to FFP arg1.

SPSqrt() `FLOAT SPSqrt( FLOAT parm );`
Return square root of FFP variable.

SPTieee() `FLOAT SPTieee( FLOAT parm );`
Convert FFP variable to IEEE format

SPFieee() `FLOAT SPFieee( FLOAT parm );`
Convert IEEE variable to FFP format.

Be sure to include proper data type definitions, as shown in the example below.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>
#include <clib/mathtrans_protos.h>

struct Library *MathTransBase;

VOID main()
{
FLOAT f1, f2, f3;
FLOAT i1;

if (MathTransBase = OpenLibrary("mathtrans.library",33))
    {
    f1 = SPAsin(f2);        /* Call SPAsin entry */
    f1 = SPAcos(f2);        /* Call SPAcos entry */
    f1 = SPAtan(f2);        /* Call SPAtan entry */

    f1 = SPSin(f2);         /* Call SPSin entry */
    f1 = SPCos(f2);         /* Call SPCos entry */
    f1 = SPTan(f2);         /* Call SPTan entry */
    f1 = SPSincos(&f3, f2); /* Call SPSincos entry */

    f1 = SPSinh(f2);        /* Call SPSinh entry */
    f1 = SPCosh(f2);        /* Call SPCosh entry */
    f1 = SPTanh(f2);        /* Call SPTanh entry */

    f1 = SPExp(f2);         /* Call SPExp entry */
    f1 = SPLog(f2);         /* Call SPLog entry */
    f1 = SPLog10(f2);       /* Call SPLog10 entry */
    f1 = SPPow(f2);         /* Call SPPow entry */
    f1 = SPSqrt(f2);        /* Call SPSqrt entry */

    i1 = SPTieee(f2);       /* Call SPTieee entry */
    f1 = SPFieee(i1);       /* Call SPFieee entry */

    CloseLibrary(MathTransBase);
    }
else
    printf("Can't open mathtrans.library\n");
}
```

The Amiga assembly language interface to the FFP transcendental math routines is shown below, including some details about how the system flags are affected by the operation. This interface resides in the library file *amiga.lib* and must be linked with the user code. Note that the access mechanism from assembly language is:

```
        MOVEA.L _MathTransBase,A6
        JSR     _LVOSPAsin(A6)
```

FFP Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOSPAsin	D0 = FFP argument	D0 = FFP arcsine radian	N = 0 Z = 1 if result is zero V = 0 C = undefined X = undefined
_LVOSPAcos	D0 = FFP argument	D0 = FFP arccosine radian	N = 0 Z = 1 if result is zero V = 1 if overflow occurred C = undefined X = undefined
_LVOSPAtan	D0 = FFP argument	D0 = FFP arctangent radian	N = 0 Z = 1 if result is zero V = 0 C = undefined X = undefined
_LVOSPSin	D0 = FFP argument in radians	D0 = FFP sine	N = 1 if result is negative Z = 1 if result is zero V = 1 if result is meaningless (that is, input magnitude too large) C = undefined X = undefined
_LVOSPCos	D0 = FFP argument in radians	D0 = FFP cosine	N = 1 if result is negative Z = 1 if result is zero V = 1 if result is meaningless (that is, input magnitude too large) C = undefined X = undefined
_LVOSPTan	D0 = FFP argument in radians	D0 = FFP tangent	N = 1 if result is negative Z = 1 if result is zero V = 1 if result is meaningless (that is, input magnitude too large) C = undefined X = undefined
_LVOSPSincos	D0 = FFP argument in radians D1 = Address to store cosine result	D0 = FFP sine (D1) = FFP cosine	N = 1 if result is negative Z = 1 if result is zero V = 1 if result is meaningless (that is, input magnitude too large) C = undefined X = undefined

FFP Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOSPSinh	D0 = FFP argument in radians	D0 = FFP hyperbolic sine	N = 1 if result is negative Z = 1 if result is zero V = 1 if overflow occurred C = undefined X = undefined
_LVOSPCosh	D0 = FFP argument in radians	D0 = FFP hyperbolic cosine	N = 1 if result is negative Z = 1 if result is zero V = 1 if overflow occurred C = undefined X = undefined
_LVOSPTanh	D0 = FFP argument in radians	D0 = FFP hyperbolic tangent	N = 1 if result is negative Z = 1 if result is zero V = 1 if overflow occurred C = undefined X = undefined
_LVOSPExp	D0 = FFP argument	D0 = FFP exponential	N = 0 Z = 1 if result is zero V = 1 if overflow occurred C = undefined Z = undefined
_LVOSPLog	D0 = FFP argument	D0 = FFP natural logarithm	N = 1 if result is negative Z = 1 if result is zero V = 1 if argument negative or zero C = undefined Z = undefined
_LVOSPLog10	D0 = FFP argument	D0 = FFP logarithm (base 10)	N = 1 if result is negative Z = 1 if result is zero V = 1 if argument negative or zero C = undefined Z = undefined
_LVOSPPow	D0 = FFP exponent value D1 = FFP argument value	D0 = FFP result of arg taken to exp power	N = 0 Z = 1 if result is zero V = 1 if result overflowed or arg < 0 C = undefined Z = undefined
_LVOSPSqrt	D0 = FFP argument	D0 = FFP square root	N = 0 Z = 1 if result is zero V = 1 if argument was negative C = undefined Z = undefined

FFP Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOSPTieee	D0 = FFP format argument	D0 = IEEE floating-point format	N = 1 if result is negative Z = 1 if result is zero V = undefined C = undefined Z = undefined
_LVOSPFieee	D0 = IEEE floating-point format argument	D0 = FFP format	N = undefined Z = 1 if result is zero V = 1 if result overflowed FFP format C = undefined Z = undefined

FFP Mathematics Conversion Library

The FFP mathematics conversion library provides functions to convert ASCII strings to their FFP equivalents and vice versa.

It is accessed by linking code into the executable file being created. The name of the file to include in the library description of the link command line is *amiga.lib*. When this is included, direct calls are made to the conversion functions. Only a C interface exists for the conversion functions; there is no assembly language interface. The basic math library is required in order to access these functions.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>

struct Library *MathBase;

VOID main()
{
if (MathBase = OpenLibrary("mathffp.library", 33))
    {
          . . .

    CloseLibrary(MathBase);
    }
else
    printf("Can't open mathffp.library\n");
}
```

MATH SUPPORT FUNCTIONS

afp() `FLOAT afp( BYTE *string );`
Convert ASCII string into FFP equivalent.

arnd() `VOID arnd( LONG place, LONG exp, BYTE *string);`
Round ASCII representation of FFP number.

dbf() `FLOAT dbf( ULONG exp, ULONG mant);`
Convert FFP dual-binary number to FFP equivalent.

fpa() `LONG fpa( FLOAT fnum, BYTE *string);`

Convert FFP variable into ASCII equivalent.

Be sure to include proper data type definitions, as shown in the example below. Print statements have been included to help clarify the format of the math conversion function calls.

```
#include <exec/types.h>
#include <libraries/mathffp.h>

#include <clib/mathffp_protos.h>
#include <clib/alib_protos.h>

struct Library *MathBase;

UBYTE st1[80] = "3.1415926535897";
UBYTE st2[80] = "2.718281828459045";
UBYTE st3[80], st4[80];

VOID main()
{
FLOAT num1, num2;
FLOAT n1, n2, n3, n4;
LONG  exp1, exp2, exp3, exp4;
LONG  mant1, mant2, mant3, mant4;
LONG  place1, place2;

if (MathBase = OpenLibrary("mathffp.library", 33))
    {

    n1 = afp(st1);            /* Call afp entry */
    n2 = afp(st2);            /* Call afp entry */
    printf("\n\nASCII %s converts to floating point %f", st1, n1);
    printf("\nASCII %s converts to floating point %f", st2, n2);

    num1 = 3.1415926535897;
    num2 = 2.718281828459045;

    exp1 = fpa(num1, st3);    /* Call fpa entry */
    exp2 = fpa(num2, st4);    /* Call fpa entry */
    printf("\n\nfloating point %f converts to ASCII %s", num1, st3);
    printf("\nfloating point %f converts to ASCII %s", num2, st4);

    place1 = -2;
    place2 = -1;
    arnd(place1, exp1, st3);    /* Call arnd entry */
    arnd(place2, exp2, st4);    /* Call arnd entry */
    printf("\n\nASCII round of %f to %d places yields %s", num1, place1, st3);
    printf("\nASCII round of %f to %d places yields %s", num2, place2, st4);

    exp1  = -3;   exp2  = 3;    exp3  = -3;   exp4  = 3;
    mant1 = 12345;  mant2 = -54321;  mant3 = -12345; mant4 = 54321;

    n1 = dbf(exp1, mant1);        /* Call dbf entry */
    n2 = dbf(exp2, mant2);        /* Call dbf entry */
    n3 = dbf(exp3, mant3);        /* Call dbf entry */
    n4 = dbf(exp4, mant4);        /* Call dbf entry */
    printf("\n\ndbf of exp = %d and mant = %d yields FFP number of %f", exp1, mant1, n1);
    printf("\ndbf of exp = %d and mant = %d yields FFP number of %f", exp2, mant2, n2);
    printf("\ndbf of exp = %d and mant = %d yields FFP number of %f", exp3, mant3, n3);
    printf("\ndbf of exp = %d and mant = %d yields FFP number of %f", exp4, mant4, n4);

    CloseLibrary(MathBase);
    }
else
    printf("Can't open mathffp.library\n");
}
```

IEEE Single-Precision Data Format

The IEEE single-precision variables are defined as 32-bit entities with the following format:

SEEEEEEE	MMMMMMMM	MMMMMMMM	MMMMMMMM
31	23	15	7

Hidden Bit In The Mantissa. There is a "hidden" bit in the mantissa part of the IEEE numbers. Since all numbers are normalized, the *integer* (high) bit of the mantissa is dropped off. The IEEE single-precision range is 1.3E-38 (1.4E-45 de-normalized) to 3.4E+38.

The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. If both the exponent and the mantissa have zero in every position, the value is zero. If only the exponent has zero in every position, the value is an *unnormal* (extremely small). If all bits of the exponent are set to 1 the value is either a positive or negative infinity or a *Not a Number (NaN).* NaN is sometimes used to indicate an uninitialized variable.

IEEE Single-Precision Basic Math Library

The ROM-based IEEE single-precision basic math library was introduced in V36. This library contains entries for the basic IEEE single-precision mathematics functions, such as add, subtract, and divide. (Note, registered developers can license a disk-based version of this library from CATS, for usage with V33).

The library is opened by making calling **OpenLibrary()** with "mathieeesingbas.library" as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter for details.

```
#include <exec/types.h>
#include <libraries/mathieeesp.h>

#include <clib/mathsingbas_protos.h>

struct Library *MathIeeeSingBasBase;

VOID main()
{
    /* do not share base pointer between tasks. */
if (MathIeeeSingBasBase = OpenLibrary("mathieeesingbas.library", 37))
    {
            .
            .
            .
    CloseLibrary(MathIeeeSingBasBase);
    }
else
    printf("Can't open mathieeesingbas.library\n");
}
```

The global variable **MathIeeeSingBasBase** is used internally for all future library references.

If an 680X0/68881/68882 processor combination is available, it will be used by the IEEE single-precision basic library instead of the software emulation. Also, if an autoconfigured math resource is available, that will be used. Typically this is a 68881 designed as a 16 bit I/O port, but it could be another device as well.

SP IEEE BASIC FUNCTIONS (V36 or greater)

IEEESPAbs() `FLOAT ( FLOAT parm );`
Take absolute value of IEEE single-precision variable.

IEEESPAdd() `FLOAT IEEESPAdd( FLOAT leftParm, FLOAT rightParm);`
Add two IEEE single-precision variables.

IEEESPCeil() `FLOAT IEEESPCeil( FLOAT parm );`
Compute least integer greater than or equal to variable.

IEEESPCmp() `LONG IEEESPCmp( FLOAT leftParm, FLOAT rightParm );`
Compare two IEEE single-precision variables.

IEEESPDiv() `FLOAT IEEESPDiv( FLOAT dividend, FLOAT divisor );`
Divide two IEEE single-precision variables.

IEEESPFix() `LONG IEEESPFix( FLOAT parm );`
Convert IEEE single-precision variable to integer.

IEEESPFloor() `FLOAT IEEESPFloor( FLOAT parm );`
Compute largest integer less than or equal to variable.

IEEESPFlt() `FLOAT IEEESPFlt( long integer );`
Convert integer variable to IEEE single-precision.

IEEESPMul() `FLOAT IEEESPMul( FLOAT leftParm, FLOAT rightParm );`
Multiply two IEEE single-precision variables.

IEEESPNeg() `FLOAT IEEESPNeg( FLOAT parm );`
Take two's complement of IEEE single-precision variable.

IEEESPSub() `FLOAT IEEESPSub( FLOAT leftParm, FLOAT rightParm );`
Subtract two IEEE single-precision variables.

IEEESPTst()F `LONG IEEESPTst( FLOAT parm );`
Test an IEEE single-precision variable against zero.

Be sure to include proper data type definitions, as shown in the example below.

```
#include <exec/types.h>
#include <libraries/mathieeesp.h>

#include <clib/mathsingbas_protos.h>

struct Library *MathIeeeSingBasBase;

VOID main()
{
FLOAT f1, f2, f3;
LONG   i1;

if (MathIeeeSingBasBase = OpenLibrary("mathieeesingbas.library",37))
    {
    i1 = IEEESPFix(f1);                /* Call IEEESPFix entry */
    fi = IEEESPFlt(i1);                /* Call IEEESPFlt entry */
    switch (IEEESPCmp(f1, f2)) {};     /* Call IEEESPCmp entry */
    switch (IEEESPTst(f1)) {};         /* Call IEEESPTst entry */
    f1 = IEEESPAbs(f2);                /* Call IEEESPAbs entry */
    f1 = IEEESPNeg(f2);                /* Call IEEESPNeg entry */
```

```
    f1 = IEEESPAdd(f2, f3);          /* Call IEEESPAdd entry */
    f1 = IEEESPSub(f2, f3);          /* Call IEEESPSub entry */
    f1 = IEEESPMul(f2, f3);          /* Call IEEESPMul entry */
    f1 = IEEESPDiv(f2, f3);          /* Call IEEESPDiv entry */
    f1 = IEEESPCeil(f2);             /* Call IEEESPCeil entry */
    f1 = IEEESPFloor(f2);            /* Call IEEESPFloor entry */

    CloseLibrary(MathIeeeSingBasBase);
    }
else
    printf("Can't open mathieeesingbas.library\n");
}
```

The Amiga assembly language interface to the IEEE single-precision basic math routines is shown below, including some details about how the system flags are affected by each operation. Note that the access mechanism from assembly language is as shown below:

```
MOVEA.L _MathIeeeSingBasBase,A6
JSR     _LVOIEEESPFix(A6)
```

SP IEEE Basic Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEESPFix	D0 = IEEE double-precision argument	D0 = Integer (two's complement)	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPFlt	D0 = Integer argument (two's complement)	D0 = IEEE single-precision	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPCmp	D0 = IEEE single-precision argument 1 D1 = IEEE single-precision argument 2	D0 = +1 if arg1 > arg2 D0 = -1 if arg1 < arg2 D0 = 0 if arg1 = arg2	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined GT = arg2 > arg1 GE = arg2 >= arg1 EQ = arg2 = arg1 NE = arg2 <> arg1 LT = arg2 < arg1 E= arg2 <= arg1
_LVOIEEESPTst	D0 = IEEE single-precision argument	D0 = +1 if arg > 0.0 D0 = -1 if arg < 0.0 D0 = 0 if arg = 0.0	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined EQ = arg = 0.0 NE = arg <> 0.0 PL = arg >= 0.0 MI = arg < 0.0

SP IEEE Basic Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEESPAbs	D0 = IEEE single-precision argument	D0 = IEEE single-precision absolute value	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPNeg	D0 = IEEE single-precision argument	D0 = IEEE single-precision negated	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPAdd	D0 = IEEE single-precision argument 1 D1 = IEEE single-precision argument 2	D0 = IEEE single-precision addition of arg1+arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPSub	D0 = IEEE single-precision argument 1 D1 = IEEE single-precision argument 2	D0 = IEEE single-precision subtraction of arg1-arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPMul	D0 = IEEE single-precision argument 1 D1 = IEEE single-precision argument 2	D0 = IEEE single-precision multiplication of arg1*arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPDiv	D0 = IEEE single-precision argument 1 D1 = IEEE single-precision argument 2	D0 = IEEE single-precision division of arg1/arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPCeil	D0 = IEEE single-precision variable	D0 = least integer >= variable	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPFloor	D0 = IEEE single-precision variable	D0 = largest integer <= argument	N = undefined Z = undefined V = undefined C = undefined X = undefined

IEEE Single-Precision Transcendental Math Library

The IEEE single-precision transcendental math library was introduced in V36. It contains entries for transcendental math functions such as sine, cosine, and square root.

This library resides on disk and is opened by calling **OpenLibrary()** with "mathieeesingtrans.library" as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter.

```
#include <exec/types.h>
#include <libraries/mathieeesp.h>

struct Library *MathIeeeSingTransBase;

#include <clib/mathsingtrans_protos.h>

VOID main()
{
if (MathIeeeSingTransBase = OpenLibrary("mathieeesingtrans.library",37))
    {
          . . .

    CloseLibrary(MathIeeeSingTransBase);
    }
else  printf("Can't open mathieeesingtrans.library\n");
}
```

The global variable **MathIeeeSingTransBase** is used internally for all future library references.

The IEEE single-precision transcendental math library is dependent upon the IEEE single-precision basic math library, which it will open if it is not open already. If you want to use the IEEE single-precision basic math functions in conjunction with the transcendental math functions however, you have to specifically open the basic math library yourself.

Just as the IEEE single-precision basic math library, the IEEE single-precision transcendental math library will take advantage of a 680X0/68881 combination or another math resource, if present.

SP IEEE TRANSCENDENTAL FUNCTIONS (V36 or greater)

IEEESPAsin() `FLOAT IEEESPAsin( FLOAT parm );`
Return arcsine of IEEE single-precision variable.

IEEESPAcos() `FLOAT IEEESPAcos( FLOAT parm );`
Return arccosine of IEEE single-precision variable.

IEEESPAtan() `FLOAT IEEESPAtan( FLOAT parm );`
Return arctangent of IEEE single-precision variable.

IEEESPSin() `FLOAT IEEESPSin( FLOAT parm );`
Return sine of IEEE single-precision variable. This function accepts an IEEE radian argument and returns the trigonometric sine value.

IEEESPCos() `FLOAT IEEESPCos( FLOAT parm );`
Return cosine of IEEE single-precision variable. This function accepts an IEEE radian argument and returns the trigonometric cosine value.

IEEESPTan() `FLOAT IEEESPTan( FLOAT parm );`
Return tangent of IEEE single-precision variable. This function accepts an IEEE radian argument and returns the trigonometric tangent value.

IEEESPSincos() `FLOAT IEEESPSincos( FLOAT *cosptr, FLOAT parm );`
Return sine and cosine of IEEE single-precision variable. This function accepts an IEEE radian argument and returns the trigonometric sine as its result and the cosine in the first parameter.

IEEESPSinh() `FLOAT IEEESPSinh( FLOAT parm );`
Return hyperbolic sine of IEEE single-precision variable.

IEEESPCosh() `FLOAT IEEESPCosh( FLOAT parm );`
Return hyperbolic cosine of IEEE single-precision variable.

IEEESPTanh() `FLOAT IEEESPTanh( FLOAT parm );`
Return hyperbolic tangent of IEEE single-precision variable.

IEEESPExp() `FLOAT IEEESPExp( FLOAT parm );`
Return *e* to the IEEE variable power. This function accept an IEEE single-precision argument and returns the result representing the value of *e* (2.712828...) raised to that power.

IEEESPFieee() `FLOAT IEEESPFieee( FLOAT parm );`
Convert IEEE single-precision number to IEEE single-precision number. The only purpose of this function is to provide consistency with the double-precision math IEEE library.

IEEESPLog() `FLOAT IEEESPLog( FLOAT parm );`
Return natural log (base *e* of IEEE single-precision variable.

IEEESPLog10() `FLOAT IEEESPLog10( FLOAT parm );`
Return log (base 10) of IEEE single-precision variable.

IEEESPPow() `FLOAT IEEESPPow( FLOAT exp, FLOAT arg );`
Return IEEE single-precision arg2 to IEEE single-precision arg1.

IEEESPSqrt() `FLOAT IEEESPSqrt( FLOAT parm );`
Return square root of IEEE single-precision variable.

IEEESPTieee() `FLOAT IEEESPTieee( FLOAT parm );`
Convert IEEE single-precision number to IEEE single-precision number. The only purpose of this function is to provide consistency with the double-precision math IEEE library.

Be sure to include the proper data type definitions as shown below.

```
#include <exec/types.h>
#include <libraries/mathieeesp.h>

#include <clib/mathsingtrans_protos.h>

struct Library *MathIeeeSingTransBase;

VOID main()
{
FLOAT f1, f2, f3;

if (MathIeeeSingTransBase = OpenLibrary("mathieeesingtrans.library",37))
    {
    f1 = IEEEDPAsin(f2);        /* Call IEEESPAsin entry */
    f1 = IEEEDPAcos(f2);        /* Call IEEESPAcos entry */
    f1 = IEEEDPAtan(f2);        /* Call IEEESPAtan entry */
    f1 = IEEEDPSin(f2);         /* Call IEEESPSin entry */
    f1 = IEEEDPCos(f2);         /* Call IEEESPCos entry */
```

```
    f1 = IEEEDPTan(f2);          /* Call IEEESPTan entry */
    f1 = IEEEDPSincos(&f3, f2);  /* Call IEEESPSincos entry */
    f1 = IEEEDPSinh(f2);         /* Call IEEESPSinh entry */
    f1 = IEEEDPCosh(f2);         /* Call IEEESPCosh entry */
    f1 = IEEEDPTanh(f2);         /* Call IEEESPTanh entry */
    f1 = IEEEDPExp(f2);          /* Call IEEESPExp entry */
    f1 = IEEEDPLog(f2);          /* Call IEEESPLog entry */
    f1 = IEEEDPLog10(f2);        /* Call IEEESPLog10 entry */
    f1 = IEEEDPPow(d2, f3);      /* Call IEEESPPow entry */
    f1 = IEEEDPSqrt(f2);         /* Call IEEESPSqrt entry */

    CloseLibrary(MathIeeeSingTransBase);
    }
else printf("Can't open mathieeesingtrans.library\n");
}
```

The section below describes the Amiga assembly interface to the IEEE single-precision transcendental math library. The access mechanism from assembly language is:

```
MOVEA.L _MathIeeeSingTransBase,A6
JSR     _LVOIEEESPAsin(A6)
```

SP IEEE Transcendental Assembly Functions

Function	Input	Output	Condition Codes
_LVOIEEESPAsin	D0 = IEEE argument	D0 = IEEE arcsine radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPAcos	D0 = IEEE single-precision argument	D0 = IEEE arccosine radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPAtan	D0 = IEEE single-precision argument	D0 = IEEE arctangent radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPSin	D0 = IEEE single-precision argument in radians	D0 = IEEE sine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPCos	D0 = IEEE single-precision argument in radians	D0 = IEEE cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPTan	D0 = IEEE single-precision argument in radians	D0 = IEEE tangent	N = undefined Z = undefined V = undefined C = undefined X = undefined

SP IEEE Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEESPSincos	A0 = Address to store cosine result D0 = IEEE argument in radians	D0 = IEEE sine (A0) = IEEE cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPSinh	D0 = IEEE single-precision argument in radians	D0 = IEEE hyperbolic sine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPCosh	D0 = IEEE single-precision argument in radians	D0 = IEEE hyperbolic cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPTanh	D0 = IEEE single-precision argument in radians	D0 = IEEE hyperbolic tangent	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPExp	D0 = IEEE single-precision argument	D0 = IEEE exponential	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPLog	D0 = IEEE single-precision argument	D0 = IEEE natural logarithm	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPLog10	D0 = IEEE single-precision argument	D0 = IEEE logarithm (base 10)	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPPow	D0 = IEEE single-precision exponent value D1 = IEEE single-precision argument value	D0 = IEEE result of arg taken to exp power	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEESPSqrt	D0 = IEEE single-precision argument	D0 = IEEE square root	N = undefined Z = undefined V = undefined C = undefined X = undefined

IEEE Double-Precision Data Format

The IEEE double-precision variables are defined as 64-bit entities with the following format:

SEEEEEEE	EEEEEIMM	MMMMMMMM	MMMMMMMM
63	55	47	39

MMMMMMMMM	MMMMMMMM	MMMMMMMM	MMMMMMMM
31	23	15	7

Hidden Bit In The Mantissa. There is a "hidden" bit in the mantissa part of the IEEE numbers. Since all numbers are normalized, the *integer* (high) bit of the mantissa is dropped off. The IEEE double-precision range is 2.2E-308 (4.9E-324 de-normalized) to 1.8E+307.

The exponent is the power of two needed to correctly position the mantissa to reflect the number's true arithmetic value. If both the exponent and the mantissa have zero in every position, the value is zero. If only the exponent has zero in every position, the value is an *unnormal* (extremely small). If all bits of the exponent are set to 1 the value is either a positive or negative infinity or a *Not a Number (NaN).* NaN is sometimes used to indicate an uninitialized variable.

IEEE Double-Precision Basic Math Library

The IEEE double-precision basic math library contains entries for the basic IEEE mathematics functions, such as add, subtract, and divide. This library resides on disk and is opened by calling **OpenLibrary()** with "mathieeedoubbas.library" as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter for details.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

#include <clib/mathdoubbas_protos.h>

struct Library *MathIeeeDoubBasBase;

VOID main()
{
    /* do not share base pointer between tasks. */
if (MathIeeeDoubBasBase = OpenLibrary("mathieeedoubbas.library", 34))
    {
        . . .

    CloseLibrary(MathIeeeDoubBasBase);
    }
else printf("Can't open mathieeedoubbas.library\n");
}
```

The global variable **MathIeeeDoubBasBase** is used internally for all future library references.

If an 680X0/68881/68882 processor combination is available, it will be used by the IEEE basic library instead of the software emulation. Also, if an autoconfigured math resource is available, that will be used. Typically this is a 68881 designed as a 16 bit I/O port, but it could be another device as well.

DP IEEE BASIC FUNCTIONS

IEEEDPAbs() `DOUBLE IEEEDPAbs( DOUBLE parm );`
Take absolute value of IEEE double-precision variable.

IEEEDPAdd() `DOUBLE IEEEDPAdd( DOUBLE leftParm, DOUBLE rightParm );`
Add two IEEE double-precision variables.

IEEEDPCeil() `DOUBLE IEEEDPCeil( DOUBLE parm );`
Compute least integer greater than or equal to variable.

IEEEDPCmp() `LONG IEEEDPCmp( DOUBLE leftParm, DOUBLE rightParm );`
Compare two IEEE double-precision variables.

IEEEDPDiv() `DOUBLE IEEEDPDiv( DOUBLE dividend, DOUBLE divisor );`
Divide two IEEE double-precision variables.

IEEEDPFix() `LONG IEEEDPFix( DOUBLE parm );`
Convert IEEE double-precision variable to integer.

IEEEDPFloor() `DOUBLE IEEEDPFloor( DOUBLE parm );`
Compute largest integer less than or equal to variable.

IEEEDPFlt() `DOUBLE IEEEDPFlt( long integer );`
Convert integer variable to IEEE double-precision.

IEEEDPMul() `DOUBLE IEEEDPMul( DOUBLE factor1, DOUBLE factor2 );`
Multiply two IEEE double-precision variables.

IEEEDPNeg() `DOUBLE IEEEDPNeg( DOUBLE parm );`
Take two's complement of IEEE double-precision variable.

IEEEDPSub() `DOUBLE IEEEDPSub( DOUBLE leftParm, DOUBLE rightParm );`
Subtract two IEEE double-precision variables.

IEEEDPTst() `LONG IEEEDPTst( DOUBLE parm );`
Test an IEEE double-precision variable against zero.

Be sure to include proper data type definitions, as shown in the example below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

#include <clib/mathieeedoubbas_protos.h>

struct Library *MathIeeeDoubBasBase;

VOID main()
{
DOUBLE d1, d2, d3;
LONG   i1;

if (MathIeeeDoubBasBase = OpenLibrary("mathieeedoubbas.library",34))
    {

    i1 = IEEEDPFix(d1);                /* Call IEEEDPFix entry */
    fi = IEEEDPFlt(i1);                /* Call IEEEDPFlt entry */
    switch (IEEEDPCmp(d1, d2)) {};     /* Call IEEEDPCmp entry */
    switch (IEEEDPTst(d1)) {};         /* Call IEEEDPTst entry */
    d1 = IEEEDPAbs(d2);                /* Call IEEEDPAbs entry */
    d1 = IEEEDPNeg(d2);                /* Call IEEEDPNeg entry */
    d1 = IEEEDPAdd(d2, d3);            /* Call IEEEDPAdd entry */
    d1 = IEEEDPSub(d2, d3);            /* Call IEEEDPSub entry */
```

```
    d1 = IEEEDPMul(d2, d3);          /* Call IEEEDPMul entry */
    d1 = IEEEDPDiv(d2, d3);          /* Call IEEEDPDiv entry */
    d1 = IEEEDPCeil(d2);             /* Call IEEEDPCeil entry */
    d1 = IEEEDPFloor(d2);            /* Call IEEEDPFloor entry */

    CloseLibrary(MathIeeeDoubBasBase);
    }
else printf("Can't open mathieeedoubbas.library\n");
}
```

The Amiga assembly language interface to the IEEE double-precision floating-point basic math routines is shown below, including some details about how the system flags are affected by each operation. The access mechanism from assembly language is:

```
        MOVEA.L _MathIeeeDoubBasBase,A6
        JSR     _LVOIEEEDPFix(A6)
```

DP IEEE Basic Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEEDPFix	D0/D1 = IEEE double-precision argument	D0 = Integer (two's complement)	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPFl	D0 = Integer (two's complement) argument	D0/D1 = IEEE double-precision	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPCmp	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2	D0 = +1 if arg1 > arg2 D0 = -1 if arg1 < arg2 D0 = 0 if arg1 = arg2	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined GT = arg2 > arg1 GE = arg2 >= arg1 EQ = arg2 = arg1 NE = arg2 <> arg1 LT = arg2 < arg1 LE = arg2 <= arg1
_LVOIEEEDPTst	D0/D1 = IEEE double-precision argument	D0 = +1 if arg > 0.0 D0 = -1 if arg < 0.0 D0 = 0 if arg = 0.0	N = 1 if result is negative Z = 1 if result is zero V = 0 C = undefined X = undefined EQ = arg = 0.0 NE = arg <> 0.0 PL = arg >= 0.0 MI = arg < 0.0

DP IEEE Basic Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEEDPAbs	D0/D1 = IEEE double-precision argument	D0/D1 = IEEE double-precision absolute value	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPNeg	D0/D1 = IEEE double-precision argument	D0/D1 = IEEE double-precision negated	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPAdd	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2	D0/D1 = IEEE double-precision addition of arg1+arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPSub	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2	D0/D1 = IEEE double-precision subtraction of arg1-arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPMul	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2	D0/D1 = IEEE double-precision multiplication of arg1*arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPDiv	D0/D1 = IEEE double-precision argument 1 D2/D3 = IEEE double-precision argument 2	D0/D1 = IEEE double-precision division of arg1/arg2	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPCeil	D0/D1 = IEEE double-precision argument	D0/D1 = least integer >= argument	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPFloor	D0/D1 = IEEE double-precision argument	D0/D1 = largest integer <= argument	N = undefined Z = undefined V = undefined C = undefined X = undefined

IEEE Double-Precision Transcendental Math Library

The IEEE double-precision transcendental math library contains entries for the transcendental math functions such as sine, cosine, and square root. The library resides on disk and is opened by calling **OpenLibrary()** with "mathieeedoubtrans.library" as the argument. Do not share the library base pointer between tasks — see note at beginning of chapter for details.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>

#include <clib/mathdoubtrans_protos.h>

struct Library *MathIeeeDoubTransBase;

VOID main()
{
if (MathIeeeDoubTransBase = OpenLibrary("mathieeedoubtrans.library",34))
    {
          . . .

    CloseLibrary(MathIeeeDoubTransBase);
    }
else printf("Can't open mathieeedoubtrans.library\n");
}
```

The global variable **MathIeeeDoubTransBase** is used internally for all future library references.

The IEEE double-precision transcendental math library is dependent upon the IEEE double-precision basic math library, which it will open if it is not open already. If you want to use the IEEE double-precision basic math functions in conjunction with the transcendental math functions however, you have to specifically open the basic math library yourself.

Just as the IEEE double-precision basic math library, the IEEE double-precision transcendental math library will take advantage of a 680X0/68881 combination or another math resource, if present.

DP IEEE TRANSCENDENTAL FUNCTIONS

IEEEDPAsin() `DOUBLE IEEEDPAsin( DOUBLE parm );`
Return arcsine of IEEE variable.

IEEEDPAcos() `DOUBLE IEEEDPAcos( DOUBLE parm );`
Return arccosine of IEEE variable.

IEEEDPAtan() `DOUBLE IEEEDPAtan( DOUBLE parm );`
Return arctangent of IEEE variable.

IEEEDPSin() `DOUBLE IEEEDPSin( DOUBLE parm );`
Return sine of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric sine value.

IEEEDPCos() `DOUBLE IEEEDPCos( DOUBLE parm );`
Return cosine of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric cosine value.

IEEEDPTan() `DOUBLE IEEEDPTan( DOUBLE parm );`
Return tangent of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric tangent value.

IEEEDPSincos() `DOUBLE IEEEDPSincos( DOUBLE *pf2, DOUBLE parm );`
Return sine and cosine of IEEE variable. This function accepts an IEEE radian argument and returns the trigonometric sine as its result and the trigonometric cosine in the first parameter.

IEEEDPSinh() `DOUBLE IEEEDPSinh( DOUBLE parm );`
Return hyperbolic sine of IEEE variable.

IEEEDPCosh() `DOUBLE IEEEDPCosh( DOUBLE parm );`
Return hyperbolic cosine of IEEE variable.

IEEEDPTanh() `DOUBLE IEEEDPTanh( DOUBLE parm );`
Return hyperbolic tangent of IEEE variable.

IEEEDPExp() `DOUBLE IEEEDPExp( DOUBLE parm );`
Return *e* to the IEEE variable power. This function accept an IEEE argument and returns the result representing the value of *e* (2.712828...) raised to that power.

IEEEDPFieee() `DOUBLE IEEEDPFieee( FLOAT single );`
Convert IEEE single-precision number to IEEE double-precision number.

IEEEDPLog() `DOUBLE IEEEDPLog( DOUBLE parm );`
Return natural log (base *e* of IEEE variable.

IEEEDPLog10() `DOUBLE IEEEDPLog10( DOUBLE parm );`
Return log (base 10) of IEEE variable.

IEEEDPPow() `DOUBLE IEEEDPPow( DOUBLE exp, DOUBLE arg );`
Return IEEE arg2 to IEEE arg1.

IEEEDPSqrt() `DOUBLE IEEEDPSqrt( DOUBLE parm );`
Return square root of IEEE variable.

IEEEDPTieee() `FLOAT IEEEDPTieee( DOUBLE parm );`
Convert IEEE double-precision number to IEEE single-precision number.

Be sure to include proper data type definitions as shown below.

```
#include <exec/types.h>
#include <libraries/mathieeedp.h>
#include <clib/mathdoubtrans_protos.h>

struct Library *MathIeeeDoubTransBase;

VOID main()
{
DOUBLE d1, d2, d3;
FLOAT f1;

if (MathIeeeDoubTransBase = OpenLibrary("mathieeedoubtrans.library",34))
    {
    d1 = IEEEDPAsin(d2);         /* Call IEEEDPAsin entry */
    d1 = IEEEDPAcos(d2);         /* Call IEEEDPAcos entry */
    d1 = IEEEDPAtan(d2);         /* Call IEEEDPAtan entry */
    d1 = IEEEDPSin(d2);          /* Call IEEEDPSin entry */
    d1 = IEEEDPCos(d2);          /* Call IEEEDPCos entry */
    d1 = IEEEDPTan(d2);          /* Call IEEEDPTan entry */
    d1 = IEEEDPSincos(&d3, d2); /* Call IEEEDPSincos entry */
    d1 = IEEEDPSinh(d2);         /* Call IEEEDPSinh entry */
```

```
    d1 = IEEEDPCosh(d2);         /* Call IEEEDPCosh entry */
    d1 = IEEEDPTanh(d2);         /* Call IEEEDPTanh entry */
    d1 = IEEEDPExp(d2);          /* Call IEEEDPExp entry */
    d1 = IEEEDPLog(d2);          /* Call IEEEDPLog entry */
    d1 = IEEEDPLog10(d2);        /* Call IEEEDPLog10 entry */
    d1 = IEEEDPPow(d2, d3);      /* Call IEEEDPPow entry */
    d1 = IEEEDPSqrt(d2);         /* Call IEEEDPSqrt entry */
    f1 = IEEEDPTieee(d2);        /* Call IEEEDPTieee entry */
    d1 = IEEEDPFieee(f1);        /* Call IEEEDPFieee entry */

    CloseLibrary(MathIeeeDoubTransBase);
    }
else printf("Can't open mathieeedoubtrans.library\n");
}
```

The section below describes the Amiga assembly interface to the IEEE double-precision transcendental math library. The access mechanism from assembly language is:

```
MOVEA.L _MathIeeeDoubTransBase,A6
JSR     _LVOIEEEDPAsin(A6)
```

DP IEEE Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEEDPAsin	D0/D1 = IEEE argument	D0/D1 = IEEE arcsine radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPAcos	D0/D1 = IEEE argument	D0/D1 = IEEE arccosine radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPAtan	D0/D1 = IEEE argument	D0/D1 = IEEE arctangent radian	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPSin	D0/D1 = IEEE argument in radians	D0/D1 = IEEE sine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPCos	D0/D1 = IEEE argument in radians	D0/D1 = IEEE cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPTan	D0/D1 = IEEE argument in radians	D0/D1 = IEEE tangent	N = undefined Z = undefined V = undefined C = undefined X = undefined

DP IEEE Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEEDPSincos	A0 = Address to store cosine result D0/D1 = IEEE argument in radians	D0/D1 = IEEE sine (A0) = IEEE cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPSin	D0/D1 = IEEE argument in radians	D0/D1 = IEEE hyperbolic sine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPCosh	D0/D1 = IEEE argument in radians	D0/D1 = IEEE hyperbolic cosine	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPTanh	D0/D1 = IEEE argument in radians	D0/D1 = IEEE hyperbolic tangent	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPExp	D0/D1 = IEEE argument	D0/D1 = IEEE exponential	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPLog	D0/D1 = IEEE argument	D0/D1 = IEEE natural logarithm	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPLog10	D0/D1 = IEEE argument	D0/D1 = IEEE logarithm (base 10)	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPPow	D0/D1 = IEEE exponent D2/D3 = IEEE argument	D0/D1 = IEEE result of arg taken to exp power	N = undefined Z = undefined V = undefined C = undefined X = undefined
_LVOIEEEDPSqrt	D0/D1 = IEEE argument	D0/D1 = IEEE square root	N = undefined Z = undefined V = undefined C = undefined X = undefined

DP IEEE Transcendental Assembly Functions			
Function	**Input**	**Output**	**Condition Codes**
_LVOIEEEDPTieee	D0/D1 = IEEE format argument	D0 = single-precision IEEE floating-point format	N = undefined Z = undefined V = undefined C = undefined X = undefined

Function Reference

Here's a brief summary of the functions covered in this chapter. Refer to the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for additional information.

FFP Basic Functions	
SPAbs()	Take absolute value of FFP variable
SPAdd()	Add two FFP variables
SPCeil()	Compute least integer greater than or equal to variable.
SPCmp()	Compare two FFP variables
SPDiv()	Divide two FFP variables
SPFix()	Convert FFP variable to integer
SPFloor()	Computer largest integer less than or equal to variable.
SPFlt()	Convert integer variable to FFP
SPMul()	Multiply two FFP variables
SPNeg()	Take two's complement of FFP variable
SPSub()	Subtract two FFP variables
SPTst()	Test an FFP variable against zero

FFP Transcendental Functions	
SPAcos()	Return arccosine of FFP variable.
SPAsin()	Return arcsine of FFP variable.
SPAtan()	Return arctangent of FFP variable.
SPCos()	Return cosine of FFP variable.
SPCosh()	Return hyperbolic cosine of FFP variable.
SPExp()	Return *e* to the FFP variable power.
SPFieee()	Convert IEEE variable to FFP format.
SPLog()	Return natural log (base *e*) of FFP variable.
SPLog10()	Return log (base 10) of FFP variable.
SPPow()	Return FFP arg2 to FFP arg1.
SPSin()	Return sine of FFP variable.
SPSincos()	Return sine and cosine of FFP variable.
SPSinh()	Return hyperbolic sine of FFP variable.
SPSqrt()	Return square root of FFP variable.
SPTan()	Return tangent of FFP variable.
SPTanh()	Return hyperbolic tangent of FFP variable.
SPTieee()	Convert FFP variable to IEEE format

Math Support Functions	
afp()	Convert ASCII string into FFP equivalent.
fpa()	Convert FFP variable into ASCII equivalent.
arnd()	Round ASCII representation of FFP number.
dbf()	Convert FFP dual-binary number to FFP equivalent.

SP IEEE Basic Functions (V36)	
IEEESPAbs()	Take absolute value of IEEE single-precision variable
IEEESPAdd()	Add two IEEE single-precision variables
IEEESPCeil()	Compute least integer greater than or equal to variable
IEEESPCmp()	Compare two IEEE single-precision variables
IEEESPDiv()	Divide two IEEE single-precision variables
IEEESPFix()	Convert IEEE single-precision variable to integer
IEEESPFloor()	Compute largest integer less than or equal to variable
IEEESPFlt()	Convert integer variable to IEEE single-precision
IEEESPMul()	Multiply two IEEE single-precision variables
IEEESPNeg()	Take two's complement of IEEE single-precision variable
IEEESPSub()	Subtract two IEEE single-precision variables
IEEESPTst()	Test an IEEE single-precision variable against zero

SP IEEE Transcendental Functions (V36)	
IEEESPACos()	Return arccosine of IEEE single-precision variable.
IEEESPASin()	Return arcsine of IEEE single-precision variable.
IEEESPAtan()	Return arctangent of IEEE single-precision variable.
IEEESPCos()	Return cosine of IEEE single-precision variable.
IEEESPCosh()	Return hyperbolic cosine of IEEE single-precision variable.
IEEESPExp()	Return *e* to the IEEE variable power.
IEEESPLog()	Return natural log (base *e* of IEEE single-precision variable.
IEEESPLog10()	Return log (base 10) of IEEE single-precision variable.
IEEESPPow()	Return power of IEEE single-precision variable.
IEEESPSin()	Return sine of IEEE single-precision variable.
IEEESPSincos()	Return sine and cosine of IEEE single-precision variable.
IEEESPSinh()	Return hyperbolic sine of IEEE single-precision variable.
IEEESPSqrt()	Return square root of IEEE single-precision variable.
IEEESPTan()	Return tangent of IEEE single-precision variable.
IEEESPTanh()	Return hyperbolic tangent of IEEE single-precision variable.

DP IEEE Basic Functions	
IEEEDPAbs()	Take absolute value of IEEE double-precision variable
IEEEDPAdd()	Add two IEEE double-precision variables
IEEEDPCeil()	Compute least integer greater than or equal to variable
IEEEDPCmp()	Compare two IEEE double-precision variables
IEEEDPDiv()	Divide two IEEE double-precision variables
IEEEDPFix()	Convert IEEE double-precision variable to integer
IEEEDPFloor()	Compute largest integer less than or equal to variable
IEEEDPFlt()	Convert integer variable to IEEE double-precision
IEEEDPMul()	Multiply two IEEE double-precision variables
IEEEDPNeg()	Take two's complement of IEEE double-precision variable
IEEEDPSub()	Subtract two IEEE single-precision variables
IEEEDPTst()	Test an IEEE double-precision variable against zero

DP IEEE Transcendental Functions	
IEEEDPACos()	Return arccosine of IEEE double-precision variable.
IEEEDPASin()	Return arcsine of IEEE double-precision variable.
IEEEDPAtan()	Return arctangent of IEEE double-precision variable.
IEEEDPCos()	Return cosine of IEEE double-precision variable.
IEEEDPCosh()	Return hyperbolic cosine of IEEE double-precision variable.
IEEEDPExp()	Return *e* to the IEEE variable power.
IEEEDPFieee()	Convert IEEE single-precision number to IEEE double-precision number.
IEEEDPLog()	Return natural log (base *e* of IEEE double-precision variable.
IEEEDPLog10()	Return log (base 10) of IEEE double-precision variable.
IEEEDPPow()	Return power of IEEE double-precision variable.
IEEEDPSin()	Return sine of IEEE double-precision variable.
IEEEDPSincos()	Return sine and cosine of IEEE double-precision variable.
IEEEDPSinh()	Return hyperbolic sine of IEEE double-precision variable.
IEEEDPSqrt()	Return square root of IEEE double-precision variable.
IEEEDPTan()	Return tangent of IEEE double-precision variable.
IEEEDPTanh()	Return hyperbolic tangent of IEEE double-precision variable.
IEEEDPTieee()	Convert IEEE double-precision number to IEEE single-precision number.

Compile and Link Commands for SAS C 5.10

FFP Basic, Transcendental and Math Support functions

lc -b1 -cfistq -ff -v -y <filename>.c
blink lib:c.o + <filename>.o TO <filename> LIB lib:lcmffp.lib + lib:lc.lib + lib:amiga.lib

IEEE Single-Precision and Double-Precision Basic and Transcendental Functions

lc -b1 -cfistq -fi -v -y <filename>.c
blink lib:c.o + <filename>.o TO <filename> LIB lib:lcmieee.lib + lib:lc.lib + lib:amiga.lib

Chapter 36
TRANSLATOR LIBRARY

This chapter describes the translator library which, together with the narrator device, provides the Amiga's text-to-speech capability. To fully understand how speech is produced on the Amiga, you should also read the "Narrator Device" chapter of the *Amiga ROM Kernel Reference Manual: Devices.*

The translator library provides a single function, **Translate()**, that converts an English language string into a phonetic string. You may then pass this phonetic string to the narrator device which will say the string using the Amiga's audio hardware. The two subsystems may also be used individually. You don't have to use the narrator to say the phonetic strings; you could use them instead for phonetic analysis or some other special purpose.

OPENING THE TRANSLATOR LIBRARY

To use the **Translate()** function, you must first open the translator library. Setting a global variable, **TranslatorBase**, to the value returned from the call to **OpenLibrary()** enables the Amiga linker to correctly locate the translator library:

```
struct Library *TranslatorBase;

TranslatorBase = OpenLibrary("translator.library",REVISION);
if(TranslatorBase != NULL)
    {
    /* use translator here -- library open */
    }
```

> *LIBS: must contain translator.library.* Since translator is a disk-based library, the call to **OpenLibrary()** will work only if the *LIBS:* directory contains *translator.library.*

USING THE TRANSLATE FUNCTION

Once the library is open, you can call the translate function:

```
#define BUFLEN 500

STRPTR EnglStr;              /* pointer to sample input string */
LONG EnglLen;                /* input length */
UBYTE PhonBuffer[BUFLEN];    /* place to put the translation */
LONG rtnCode;                /* return code from function */

EnglStr = "This is Amiga speaking.";     /* a test string */
EnglLen = strlen(EnglStr);
rtnCode = Translate(EnglStr, EnglLen, (STRPTR)&PhonBuffer[0], BUFLEN);
```

The input string will be translated into its phonetic equivalent and can be used to feed the narrator device. If you receive a non-zero return code, you haven't provided enough output buffer space to hold the entire translation. In this case, the **Translate()** function breaks the translation at the end of a word in the input stream and returns the position in the input stream at which the translation ended. You can use the output buffer, then call the **Translate()** function again, starting at this original ending position, to continue the translation where you left off. This method will sound smoothest if the ending position ends on sentence boundaries.

> *Translate() returns negative values.* To get the proper character offset, you must use -(**rtnCode**) as the starting point for a new translation.

CLOSING THE TRANSLATOR LIBRARY

As with all other libraries of functions, if you have successfully opened the translator library for use, be sure to close it before your program exits. If the system needs memory resources, it can then expunge closed libraries to gain additional memory space:

```
struct Library *TranslatorBase;

if(TranslatorBase) CloseLibrary(TranslatorBase);
```

ADDITIONAL NOTES ABOUT TRANSLATE

The English language has many words that do not sound the same as they are spelled. The translator library has exception rules that it consults as the translation progresses. It also provides for common abbreviations such as Dr., Prof., LB., etc. Words that are not in the exception table are translated literally. This translation allows unrestricted English text as input, and uses over four hundred and fifty context sensitive rules. It automatically accents content words, and leaves function words (e.g. of, by, the, and at) unaccented. It is possible, however, that certain words will not translate well. You can improve the quality of translation by handling those words on your own.

The phoneme table that the narrator uses is listed in the ''Narrator Device'' chapter of the *Amiga ROM Kernel Reference Manual: Devices*. You will also find other important information about the Amiga's speech capability in the narrator chapter including a working example which shows how to use the translator library together with the narrator device.

Chapter 37
UTILITY LIBRARY

Utility library is the home for all the OS functions which don't fit in the other libraries. Among the calls in utility library are calls to manage tags and tag lists, callback hooks, and some generic 32-bit math functions, all discussed below.

Tags

The implementation of tags is one of the many new features of Release 2. Tags make it possible to add new parameters to system functions without interfering with the original parameters. They also make specifying parameter lists much clearer and easier.

TAG FUNCTIONS AND STRUCTURES

A tag is made up of an attribute/value pair as defined below (from <*utility/tagitem.h*>):

```
struct TagItem
{
    ULONG  ti_Tag;     /* identifies the type of this item */
    ULONG  ti_Data;    /* type-specific data, can be a pointer */
};
```

The **ti_Tag** field specifies an attribute to set. The possible values of **ti_Tag** are implementation specific. System tags are defined in the include files. The value the attribute is set to is specified in **ti_Data**. An example of the attribute/value pair that will specify a window's name is:

```
ti_Tag  = WA_Title;
ti_Data = "My Window's Name";
```

The **ti_Data** field often contains 32-bit data as well as pointers.

These are brief descriptions of the utility functions you can use to manipulate and access tags. For complete descriptions, see the ''Simple Tag Usage'' and ''Advanced Tag Usage'' sections.

The following utility library calls are for supporting tags:

Table 37-1: Utility Library Tag Functions

AllocateTagItems()	Allocate a **TagItem** array (or chain).
FreeTagItems()	Frees allocated **TagItem** lists.
CloneTagItems()	Copies a **TagItem** list.
RefreshTagItemClone()	Rejuvenates a clone from the original.
FindTagItem()	Scans **TagItem** list for a tag.
GetTagData()	Obtain data corresponding to tag.
NextTagItem()	Iterate **TagItem** lists.
TagInArray()	Check if a tag value appears in a **Tag** array.
FilterTagChanges()	Eliminate **TagItems** which specify no change.
FilterTagItems()	Remove selected items from a **TagItem** list.
MapTags()	Convert **ti_Tag** values in a list via map pairing.
PackBoolTags()	Builds a "Flag" word from a **TagItem** list.

SIMPLE TAG USAGE

One way tags are passed to system functions is in the form of tag lists. A tag list is an array or chain of arrays of **TagItem** structures. Within this array, different data items are identified by the value of **ti_Tag**. Items specific to a subsystem (Intuition, Graphics,...) have a **ti_Tag** value which has the TAG_USER bit set. Global system tags have a **ti_Tag** value with TAG_USER bit clear. The global system tags include:

Table 37-2: Global System Tags

Tag Value	Meaning
TAG_IGNORE	A no-op. The data item is ignored.
TAG_MORE	The ti_Data points to another tag list, to support chaining of **TagItem** arrays.
TAG_DONE	Terminates the TagItem array (or chain).
TAG_SKIP	Ignore the current tag item, and skip the next n array elements, where n is kept in ti_Data.

Note that user tags need only be unique within the particular context of their use. For example, the attribute tags defined for **OpenWindow()** have the same numeric value as some tags used by **OpenScreen()**, but the same numeric value has different meaning in the different contexts.

System functions receive **TagItems** in several ways. One way is illustrated in the Intuition function **OpenWindow()**. This function supports an extented **NewWindow** structure called **ExtNewWindow**. When the NW_EXTENDED flag is set in the **ExtNewWindow.Flags** field, **OpenWindow()** assumes that the **ExtNewWindow.Extension** field contains a pointer to a tag list.

Another method of passing a tag list is to directly pass a pointer to a tag list, as **OpenWindowTagList()** does in the following code fragment.

```
struct TagItem tagitem[3];
struct Screen *screen;
struct Window *window;

tagitem[0].ti_Tag = WA_CustomScreen;
tagitem[0].ti_Data = screen;    /* Open on my own screen */
tagitem[1].ti_Tag = WA_Title;
tagitem[1].ti_Data = "RKM Test Window";
tagitem[2].ti_Tag = TAG_DONE;   /* Marks the end of the tag array. */

/* Use defaults for everything else. Will open as big as the screen. */
/* Because all window parameters are specified using tags, we don't  */
/* need a NewWindow structure                                         */

if (window = OpenWindowTagList(NULL, tagitem))
{
    /* rest of code */
    CloseWindow(window);
}
```

Notice that window parameters need not be explicitly specified. Functions that utilize tags have reasonable defaults to fall back on in case no valid attribute/value pair was supplied for a particular parameter. This fall back capability is a useful feature. An application only has to specify the attributes that differ from the default, rather than unnecessarily listing all the possible attributes.

The *amiga.lib* support library offers another way to pass **TagItems** to a function. Rather than passing a tag list, the function **OpenWindowTags()** receives the attribute/value pairs in the argument list, much like **printf()** receives its arguments. Any number of attribute/value pairs can be specified. This type of argument passing is called VarArgs. The following code fragment illustrates the usage of **OpenWindowTags()**.

```
struct Window *window;

/* Just pass NULL to show we aren't using a NewWindow */
window = OpenWindowTags( NULL,
                         WA_CustomScreen, screen,
                         WA_Title, "RKM Test Window",
                         TAG_DONE );
```

Tags are not exclusively for use with the operating system; the programmer can implement them as well. The run-time utility library contains several functions to make using tags easier.

SIMPLE TAG USAGE EXAMPLE

The following example shows simple usage of tags. It shows how to allocate a tag array and use it, it also shows how to build a tag array on the stack.

```
;/* tag1.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfis -j73 tag1.c
Blink FROM LIB:c.o,tag1.o TO tag1 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <exec/libraries.h>
#include <utility/tagitem.h>
#include <intuition/intuition.h>
#include <clib/exec_protos.h>
#include <clib/intuition_protos.h>
#include <clib/utility_protos.h>

extern struct Library *SysBase;
struct Library *IntuitionBase, *UtilityBase;

int main (int argc, char **argv)
{
    struct TagItem *tags;
    struct Window  *win;
```

```
    /* For this example we need Version 2.0 */
    if (IntuitionBase = OpenLibrary ("intuition.library", 37))
    {
        /* We need the utility library for this example */
        if (UtilityBase = OpenLibrary ("utility.library", 37))
        {

        /****************************************************************/
        /* This section allocates a tag array, fills it in with values, */
        /* and then uses it.                                            */
        /****************************************************************/

            /* Allocate a tag array */
            if (tags = AllocateTagItems (7))
            {
                /* Fill in our tag array */
                tags[0].ti_Tag = WA_Width;
                tags[0].ti_Data = 320;
                tags[1].ti_Tag = WA_Height;
                tags[1].ti_Data = 50;
                tags[2].ti_Tag = WA_Title;
                tags[2].ti_Data = (ULONG) "RKM Tag Example 1";
                tags[3].ti_Tag = WA_IDCMP;
                tags[3].ti_Data = IDCMP_CLOSEWINDOW;
                tags[4].ti_Tag = WA_CloseGadget;
                tags[4].ti_Data = TRUE;
                tags[5].ti_Tag = WA_DragBar;
                tags[5].ti_Data = TRUE;
                tags[6].ti_Tag = TAG_DONE;

                /* Open the window, using the tag attributes as the
                 * only description. */
                if (win = OpenWindowTagList (NULL, tags))
                {
                    /* Wait for an event to occur */
                    WaitPort (win->UserPort);

                    /* Close the window now that we're done with it */
                    CloseWindow (win);
                }

                /* Free the tag list now that we're done with it */
                FreeTagItems(tags);
            }

        /****************************************************************/
        /* This section builds the tag array on the stack, and passes   */
        /* the array to a function.                                     */
        /****************************************************************/

            /* Now use the VarArgs (or stack based) version. */
            if (win = OpenWindowTags ( NULL,
                                       WA_Width, 320,
                                       WA_Height, 50,
                                       WA_Title, (ULONG)"RKM Tag Example 1",
                                       WA_IDCMP, IDCMP_CLOSEWINDOW,
                                       WA_CloseGadget, TRUE,
                                       WA_DragBar, TRUE,
                                       TAG_DONE))
            {
                /* Wait for an event to occur */
                WaitPort (win->UserPort);

                /* Close the window now that we're done with it */
                CloseWindow (win);
            }

            /* Close the library now */
            CloseLibrary (UtilityBase);
        }

        /* Close the library now that we're done with it */
        CloseLibrary (IntuitionBase);
    }
}
```

ADVANCED TAG USAGE

The previous section provided the background material necessary to start using tags. This section will show how to use the more advanced features of tags using functions within utility library.

Creating a New Tag List

The **AllocateTagItems()** function can be used to create a new tag array ready for use. The tag array should be passed to **FreeTagItems()** when the application is done with it.

```
struct TagItem *tags;
ULONG tags_needed;

/* Indicate how many tags we need */
tags_needed = 10;

/* Allocate a tag array */
if (tags = AllocateTagItems(tags_needed))
{
    /* ...do something with the array... */

    /* Free the array when your done with it */
    FreeTagItems (tags);
}
```

Copying an Existing Tag List

The **CloneTagItems()** function is used to copy an existing tag array into a new tag array.

```
struct TagItem *otags;      /* Original tag array */
struct TagItem *ntags;      /* New tag array */

/* Make sure there is a TagItem array */
if (otags)
{
    /* Copy the original tags into a new tag array */
    if (ntags = CloneTagItems(otags))
    {
        /* ...do something with the array... */

        /* Free the array when your done with it */
        FreeTagItems (ntags);
    }
}
```

This function can also be used to implement a function that will insert tag items into an array.

```
struct TagItem *otags;      /* Original tag array */
struct TagItem *tags;       /* New tag array */

/* Insert a couple of tags into an existing tag array */
if (tags = MakeNewTagList (GA_LeftEdge, 10,
                           GA_TopEdge, 20,
                           TAG_MORE, otags))
{
    /* ...do something with the array... */

    /* Free the array when your done with it */
    FreeTagItems (tags);
}

/* This function will create a tag array from tag pairs placed on
 * the stack */
struct TagItem *MakeNewTagList (ULONG data,...)
{
    struct TagItem *tags = (struct TagItem *) &data;

    return (CloneTagItems (tags));
}
```

Filtering an Existing Tag List

Sometimes it is necessary to only allow certain attributes to be visible in a tag list. In order to achieve this, the tag array would need to be filtered.

A number of functions are provided for filtering items in a tag array. They are **FilterTagChanges()**, **FilterTagItems()** and **RefreshTagItemClone()**.

```
/* We want the text entry gadget to receive the following tags */
Tag string_attrs[] =
{
    STRINGA_MaxChars,
    STRINGA_Buffer,
    STRINGA_TextVal,
    TAG_END,
};

/* These are attributes that the model understands */
Tag model_attrs[] =
{
    CGTA_Total,
    CGTA_Visible,
    CGTA_Top,
    ICA_TARGET,
    ICA_MAP,
    TAG_END,
};

struct TagItem *otags;      /* Original tag list */
struct TagItem *ntags;      /* New, work, tag list */

/* Make a copy of the original for us to work with */
ntags = CloneTagItems (otags);

/* Create a tag list that only contains attributes that are
 * listed in the model_attrs list. */
if (FilterTagItems (ntags, model_attrs, TAGFILTER_AND))
{
    /* Work with filtered tag list (ntags) */

    /* Restore the tag list */
    RefreshTagItemClones (ntags, otags);

    /* Create a tag list that only contains attributes that
     * aren't in the model_attrs list. */
    if (FilterTagItems (ntags, model_attrs, TAGFILTER_NOT))
    {
        /* Work with filtered tag list (ntags) */
    }

    /* Restore the tag list */
    RefreshTagItemClones (ntags, otags);

    /* Create a tag list that only contains attributes that
     * are in the string_attrs list. */
    if (FilterTagItems (ntags, string_attrs, TAGFILTER_AND))
    {
        /* Work with filtered tag list (ntags) */
    }
}

/* Free work tag list. */
FreeTagItems (ntags);
```

Locating an Attribute

To see if an attribute is in a tag array, the TagInArray() function is used.

```
/* See if the listview labels attribute is located in a tag array */
if (TagItemArray(GTLV_Labels, tags))
{
    /* Yes, the attribute is in the list */
}
else
{
    /* No, the attribute isn't in the list */
}
```

The **FindTagItem**() function will return a pointer to the actual tag that has the desired attribute. This allows you to manipulate the tag or to determine if the attribute exists but just has a NULL value.

```
struct TagItem *tag;

/* See if they are trying to set a sound */
if (tag = FindTagItem(MGA_Sound, attrs))
{
    /* Set the sound attribute to point to the specified sound data */
    tag->ti_Data = sound;
}
```

Sequential Access of Tag Lists

In order to sequentially access the members of a tag array, the **NextTagItem**() function is used.

```
struct TagItem *tags = msg->ops_AttrList;
struct TagItem *tstate;
struct TagItem *tag;
ULONG tidata;

/* Start at the beginning */
tstate = tags;

/* Step through the tag list while there are still items in the
 * list */
while (tag = NextTagItem (&tstate))
{
    /* Cache the data for the current element */
    tidata = tag->ti_Data;

    /* Handle each attribute that we understand */
    switch (tag->ti_Tag)
    {
        /* Put a case statement here for each attribute that your
         * function understands */
        case PGA_Freedom:
            lod->lod_Flags |= tidata;
            break;

        case GTLV_Labels:
            lod->lod_List = (struct List *) tidata;
            break;

        /* We don't understand this attribute */
        default:
            break;
    }
}
```

Random Access of Tag Lists

The **GetTagData()** function will return the data for the specified attribute. If there isn't a tag that matches, then the default value is returned.

```
APTR sound;

/* Get the sound data that our function will use. */
sound = (APTR) GetTagData (MGA_Sound, (ULONG) DefaultSound, attrs);
```

Obtaining Boolean Values

Often times data is best represented as simple boolean (TRUE or FALSE) values. The **PackBoolTags()** function provides an easy method for converting a tag list to bit fields.

```
/* These are the attributes that we understand, with the
 * corresponding flag value. */
struct TagItem activation_bools[] =
{
    /* Attribute           Flags */
    {GA_ENDGADGET,         ENDGADGET},
    {GA_IMMEDIATE,         GADGIMMEDIATE},
    {GA_RELVERIFY,         RELVERIFY},
    {GA_FOLLOWMOUSE,       FOLLOWMOUSE},
    {GA_RIGHTBORDER,       RIGHTBORDER},
    {GA_LEFTBORDER,        LEFTBORDER},
    {GA_TOPBORDER,         TOPBORDER},
    {GA_BOTTOMBORDER,      BOTTOMBORDER},
    {GA_TOGGLESELECT,      TOGGLESELECT},

    /* Terminate the array */
    {TAG_END}
};

/* Set the activation field, based on the attributes passed */
g->Activation = PackBoolTags(g->Activation, tags, activation_bools);
```

Mapping Tag Attributes

To translate all occurrences of an attribute to another attribute, the **MapTags()** function is used.

For Release 2, the third parameter of this function is always TRUE (tags remain in the array even if they can't be mapped).

```
struct TagItem map_list[] =
{
    /* Original     New */
    {MGA_LeftEdge,  GA_LeftEdge},
    {MGA_TopEdge,   GA_TopEdge},
    {MGA_Width,     GA_Width},
    {MGA_Height,    GA_Height},

    /* Terminate the array */
    {TAG_END},
}

/* Map the tags to the new attributes, keeping all attributes that
 * aren't included in the mapping array */
MapTags(tags, map_list, TRUE);
```

Callback Hooks

The callback features of Release 2 provide a standard means for applications to extend the functionality of libraries, devices, and applications. This standard makes it easy for the operating system to use custom modules from different high level programming languages as part of the operating system. For example, the layers library, which takes care of treating a display as a series of layered regions, allows an application to attach a pattern function to a display layer. Instead of filling in the background of a layer with the background color, the layers library calls the custom pattern function which fills in the layer display with a custom background pattern.

CALLBACK HOOK STRUCTURE AND FUNCTION

An application passes a custom function in the form of a callback **Hook** (from *<utility/hooks.h>*):

```
struct Hook
{
    struct MinNode h_MinNode;
    ULONG (*h_Entry)();      /* stub function entry point */
    ULONG (*h_SubEntry)();   /* the custom function entry point */
    VOID *h_Data;            /* owner specific */
};
```

h_MinNode
: This field is reserved for use by the module that will call the Hook.

h_Entry
: This is the address of the Hook stub. When the OS calls a callback function, it puts parameters for the callback function in CPU registers A0, A1, and A2. This makes it tough for higher level language programmers to use a callback function because most higher level languages don't have a way to manipulate CPU registers directly. The solution is a stub function which first copies the parameters from the CPU registers to a place where a high level language function can get to them. The stub function then calls the callback function. Typically, the stub pushes the registers onto the stack in a specific order and the callback function pops them off the stack.

h_SubEntry
: This is the address of the actual callback function that the application has defined. The stub calls this function.

h_Data
: This field is for the application to use. It could point to a global storage structure that the callback function utilizes.

There is only one function defined in utility library that relates to callback hooks:

```
ULONG CallHookPkt(struct Hook *hook, VOID *object, VOID *paramPkt);
```

This function calls a standard callback Hook function.

Simple Callback Hook Usage

A Hook function must accept the following three parameters in these specific registers:

A0 Pointer to the **Hook** structure.
A2 Pointer to an object to manipulate. The object is context specific.
A1 Pointer to a message packet. This is also context specific.

For a callback function written in C, the parameters should appear in this order:

```
myCallbackFunction(Pointer to Hook (A0),
                   Pointer to Object (A2),
                   Pointer to message (A1));
```

This is because the standard C stub pushes the parameters onto the stack in the following order: A1, A2, A0. The following assembly language routine is a callback stub for C:

```
INCLUDE 'exec/types.i'
INCLUDE 'utility/hooks.i'

xdef        _hookEntry

_hookEntry:
    move.l  a1,-(sp)                ; push message packet pointer
    move.l  a2,-(sp)                ; push object pointer
    move.l  a0,-(sp)                ; push hook pointer
    move.l  h_SubEntry(a0),a0       ; fetch actual Hook entry point ...
    jsr     (a0)                    ; and call it
    lea     12(sp),sp               ; fix stack
    rts
```

If your C compiler supports registerized parameters, your callback functions can get the parameters directly from the CPU registers instead of having to use a stub to push them on the stack. The following C language routine uses registerized parameters to put parameters in the right registers. This routine requires a C compiler that supports registerized parameters.

```
#include <exec/types.h>
#include <utility/hooks.h>

#define     ASM     __asm
#define     REG(x)  register __ ## x

/* This function converts register-parameter hook calling
 * convention into standard C conventions.  It requires a C
 * compiler that supports registerized parameters, such as
 * SAS/C 5.xx or greater.
 */
ULONG ASM
hookEntry(REG(a0) struct Hook *h, REG(a2) VOID *o, REG(a1) VOID *msg)
{
    return ((*h->h_SubEntry)(h, o, msg));
}
```

A callback function is executed on the context of the module that invoked it. This usually means that callback functions cannot call functions that need to look at environment specific data. For example, **printf()** needs to look at the current process's input and output stream. Entities like Intuition have no input and output stream. This also means that in order for the function to access any of its global data, it needs to make sure the CPU can find the function's data segment. It does this by forcing the function to load the offset for the program's data segment into CPU register A4. See your compiler documentation for details.

The following is a simple function that can be used in a callback hook.

```
ULONG MyFunction (struct Hook *h, VOID *o, VOID *msg)
{
    /* A SASC and Manx function that obtains access to the global
       data segment */
    geta4();

    /* Debugging function to send a string to the serial port */
    KPrintF("Inside MyFunction()\n");

    return (1);
}
```

The next step is to initialize the **Hook** for use. This basically means that the fields of the **Hook** structure must be filled with appropriate values.

The following simple function initializes a **Hook** structure.

```
/* This simple function is used to initialize a Hook */
VOID InitHook (struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the hook fields */
        h->h_Entry = (ULONG (*)()) hookEntry;
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}
```

The following is a simple example of a callback hook function.

```
;/* hooks1.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfis -j73 hooks1.c
Blink FROM LIB:c.o,hooks1.o TO hooks1 LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

    #include <exec/types.h>
    #include <exec/libraries.h>
    #include <utility/hooks.h>
    #include <dos.h>

    #include <clib/exec_protos.h>
    #include <clib/dos_protos.h>
    #include <clib/utility_protos.h>

    #include <stdio.h>

    extern struct Library *SysBase;
    struct Library *UtilityBase;

    #define ASM      __asm
    #define REG(x)   register __ ## x

    /* This function converts register-parameter Hook calling
     * convention into standard C conventions.  It requires a C
     * compiler that supports registerized parameters, such as
     * SAS/C 5.xx or greater.
     */

    ULONG ASM
    hookEntry(REG(a0) struct Hook *h, REG(a2) VOID *o, REG(a1) VOID *msg)
    {
        return ((*(ULONG (*)(struct Hook *,VOID *,VOID *))(h->h_SubEntry))(h, o, msg));
    }
```

```
/* This simple function is used to initialize a Hook */
VOID InitHook (struct Hook *h, ULONG (*func)(), VOID *data)
{
    /* Make sure a pointer was passed */
    if (h)
    {
        /* Fill in the Hook fields */
        h->h_Entry = (ULONG (*)()) hookEntry;
        h->h_SubEntry = func;
        h->h_Data = data;
    }
}

/* This function only prints out a message indicating that we are
 * inside the callback function.
 */

ULONG MyFunction (struct Hook *h, VOID *o, VOID *msg)
{
    /* Obtain access to the global data segment */
    geta4();

    /* Debugging function to send a string to the serial port */
    printf("Inside MyFunction()\n");

    return (1);
}

int main (int argc, char **argv)
{
    struct Hook h;

    /* Open the utility library */
    if (UtilityBase = OpenLibrary ("utility.library", 36))
    {
        /* Initialize the callback Hook */
        InitHook (&h, MyFunction, NULL);

        /* Use the utility library function to invoke the Hook */
        CallHookPkt (&h, NULL, NULL);

        /* Close utility library now that we're done with it */
        CloseLibrary (UtilityBase);
    }
    else printf ("Couldn't open utility.library\n");
}
```

32-bit Integer Math Functions

Utility library contains some high-speed math functions for 32-bit integer division and multiplication. These functions will take advantage of available processor instructions (like DIVUL), if a 68020 processor or higher is present. If not, these functions will mimic those instructions in 68000 only instructions, thus providing processor independency.

Currently the following functions are implemented:

SDivMod32()	Signed 32 by 32-bit division and modulus.
SMult32()	Signed 32 by 32-bit multiplication.
UDivMod32()	Unsigned 32 by 32-bit division and modulus.
UMult32()	Unsigned 32 by 32-bit multiplication.

Table 37-3: Utility Library 32-bit Math Functions

The division functions return the quotient in D0 and the remainder in D1. To obtain the remainder in a higher level language, either a compiler specific instruction to fetch the contents of a specific register must be used (like **getreg()** in SAS C) or a small assembler stub.

Following a simple example of the usage of the 32-bit integer math functions in C.

```
;/* uptime.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfis -j73 uptime.c
Blink FROM LIB:c.o,uptime.o TO uptime LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
    /* Uses SAS C getreg() */

    #include <exec/types.h>
    #include <exec/memory.h>
    #include <dos/dos.h>
    #include <dos/dosextens.h>
    #include <dos/datetime.h>
    #include <utility/date.h>
    #include <dos.h>

    #include <clib/exec_protos.h>
    #include <clib/dos_protos.h>
    #include <clib/utility_protos.h>

    #include <stdlib.h>

    struct Library *UtilityBase;

    LONG main(void);

    LONG main(void)
    {
      struct InfoData *infodata;
      struct DeviceList *ramdevice;
      struct DateStamp *now;
      LONG currenttime, boottime;
      BPTR lock;
      LONG vargs[3];
      LONG rc = RETURN_FAIL;

      /* Fails silently if < 37 */
      if (UtilityBase = OpenLibrary("utility.library", 37))
      {
        if (infodata = AllocMem(sizeof(struct InfoData), MEMF_CLEAR))
        {
          if (now = AllocMem(sizeof(struct DateStamp), MEMF_CLEAR))
          {
            if (lock = Lock("RAM:", SHARED_LOCK))
            {
              if ((Info(lock, infodata)) == DOSTRUE)
              {
                /* Make C pointer */

                ramdevice = BADDR(infodata->id_VolumeNode);

                boottime = SMult32(ramdevice->dl_VolumeDate.ds_Days, 86400) +
                           SMult32(ramdevice->dl_VolumeDate.ds_Minute, 60) +
                           SDivMod32(ramdevice->dl_VolumeDate.ds_Tick,
                                     TICKS_PER_SECOND );

                DateStamp(now);

                currenttime = SMult32(now->ds_Days, 86400) +
                              SMult32(now->ds_Minute, 60) +
                              SDivMod32(now->ds_Tick, TICKS_PER_SECOND);

                currenttime -= boottime;

                if (currenttime > 0)
                {
                  vargs[0] = UDivMod32(currenttime, 86400);
```

```
            vargs[1] = getreg(1);
            vargs[1] = UDivMod32(vargs[1], 3600);

            vargs[2] =getreg(1);
            vargs[2] = UDivMod32(vargs[2], 60);

            /*
             * Passing the address of the array allows the VPrintf()
             * function to access the array contents.  Keep in mind
             * that VPrintf() does _NOT_ know how many elements are
             * really valid in the final parameter, and will gleefully
             * run past the valid arguments.
             */
            VFPrintf(Output(),
                     "up for %ld days, %ld hours, %ld minutes\n",
                     vargs );

            rc = RETURN_OK;
          }
        }
        UnLock(lock);
      }
      FreeMem(now, sizeof(struct DateStamp));
    }
    FreeMem(infodata, sizeof(struct InfoData));
  }
  CloseLibrary(UtilityBase);
 }
 exit(rc);
}
```

International String Functions

The international string functions in utility library are a way to make use of a future localization library which Commodore-Amiga will provide. When the localization library is opened, the functions will be replaced by ones which will take the locale as defined by the user into account. This means that the compare order may change according to the locale, so care should be taken not to rely on obtaining specific compare sequences.

Currently implemented are:

Stricmp()	Compare string case-insensitive.
Strnicmp()	Compare string case-insensitive, with a specified length.
ToLower()	Convert a character to lower case.
ToUpper()	Convert a character to upper case.

Table 37-4: Utility Library International String Functions

These functions operate in the same manner as their ANSI C equivalents, for the most part. For more information, see the "Utility Library" Autodocs in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs*. Here is a simple example of the usage of the international string functions.

```
;/* istr.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfis -j73 istr.c
Blink FROM LIB:c.o,istr.o TO istr LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

    #include <exec/types.h>
    #include <stdio.h>
    #include <string.h>

    #include <clib/exec_protos.h>
```

```
#include <clib/utility_protos.h>

void main(void);
struct Library *UtilityBase;

void main(void)
{
    UBYTE *butter = "Bxtervlxxt";
    UBYTE *bread = "Kneckerbrxt";
    UBYTE ch1, ch2;
    LONG result;

    /* Fails silently if < 37 */
    if (UtilityBase = OpenLibrary("utility.library", 37))
    {
            result = Stricmp(butter, bread);

            printf("comparing %s with %s yields %ld\n", butter, bread, result );

            result = Strnicmp(bread, butter, strlen(bread));

            printf("comparing (with length) %s with %s yields %ld\n", bread, butter, result );

            ch1 = ToUpper(0xE6); f /* ASCII character 230 ae ligature */
            ch2 = ToLower(0xD0); P /* ASCII character 208 Icelandic Eth */

            printf("Chars %c %c\n", ch1, ch2);
    }
}
```

Date Functions

To ease date-related calculations, the utility library has some functions to convert a date, specified in a **ClockData** structure, in the number of seconds since 00:00:00 01-Jan-78 and vice versa. To indicate the date, the **ClockData** structure (in *<utility/date.h>*) is used.

```
struct ClockData
{
    UWORD sec;      /* seconds (0 - 59)*/
    UWORD min;      /* minutes (0 - 59) */
    UWORD hour;     /* hour (0 - 23) */
    UWORD mday;     /* day of the month (1 - 31) */
    UWORD month;    /* month of the year (1 - 12)
    UWORD year;     /* 1978 - */
    UWORD wday;     /* day of the week (0 - 6, where 0 is Sunday) */
};
```

The following functions are available to operate on **ClockData**:

Amiga2Date()	Calculate the date from the specified timestamp (in seconds).
CheckDate()	Check the legality of a date.
Date2Amiga()	Calculate the timestamp from the specified date.

Table 37-5: Utility Library Date Functins

Amiga2Date() takes a number of seconds from 01-Jan-78 as argument and fills in the supplied **ClockData** structure with the date and time.

CheckDate() checks if the supplied **ClockData** structure is valid, and returns the number of seconds from 01-Jan-78 if it is. Note that this function currently does not take the supplied day of the week in account.

Date2Amiga() takes a **ClockData** structure as argument and returns the number of seconds since 01-Jan-78. The supplied **ClockData** structure MUST be valid, since no checking is done.

The following example shows various uses of the utility library date functions.

```
;/* a2d.c - Execute me to compile me with SAS C 5.10
LC -b1 -cfis -j73 a2d.c
Blink FROM LIB:c.o,a2d.o TO a2d LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/
    #include <exec/types.h>
    #include <exec/memory.h>
    #include <dos/datetime.h>
    #include <devices/timer.h>

    #include <clib/exec_protos.h>
    #include <clib/timer_protos.h>
    #include <clib/utility_protos.h>

    #include <stdio.h>

    LONG main(void);

    struct Library *TimerBase;
    struct Library *UtilityBase;

    LONG main(void)
    {
      struct ClockData *clockdata;
      struct timerequest *tr;
      struct timeval *tv;
      LONG seconds;

      if (UtilityBase = OpenLibrary("utility.library", 37))
      {
        if (tr = AllocMem(sizeof(struct timerequest), MEMF_CLEAR))
        {
          if (tv = AllocMem(sizeof(struct timeval), MEMF_CLEAR))
          {
            if (clockdata = AllocMem(sizeof(struct ClockData), MEMF_CLEAR))
            {
              if (!(OpenDevice("timer.device", UNIT_VBLANK, (struct IORequest *)tr, 0) ))
              {
                TimerBase = tr->tr_node.io_Device;

                GetSysTime(tv);

                printf("GetSysTime():\t%d %d\n", tv->tv_secs, tv->tv_micro);

                Amiga2Date(tv->tv_secs, clockdata);

                printf("Amiga2Date():  sec %d min %d hour %d\n", clockdata->sec,
                        clockdata->min, clockdata->hour);

                printf("               mday %d month %d year %d wday %d\n", clockdata->mday,
                       clockdata->month, clockdata->year, clockdata->wday);

                seconds = CheckDate(clockdata);

                printf("CheckDate():\t%ld\n", seconds);

                seconds = Date2Amiga(clockdata);

                printf("Date2Amiga():\t%ld\n", seconds);

                CloseDevice((struct IORequest *)tr);
              }
              FreeMem(clockdata, sizeof(struct ClockData));
            }
            FreeMem(tv, sizeof(struct timeval));
          }
          FreeMem(tr, sizeof(struct timerequest));
        }
        CloseLibrary(UtilityBase);
      }
    }
```

Function Reference

The tables which follow contain breif descriptions of the functions inside the utility library. All these functions require Release2 or a later version of the Amiga operating system. See the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for details on each function call.

TAG FUNCTION REFERENCE

The following are brief descriptions of the utility library functions which pertain to tags and tag lusts.

Table 37-6: Utility Tag Functions

Function	Description
AllocateTagItems()	Allocate a **TagItem** array (or chain).
FreeTagItems()	Frees allocated **TagItem** lists.
CloneTagItems()	Copies a **TagItem** list.
RefreshTagItemClone()	Rejuvenates a clone from the original.
FindTagItem()	Scans **TagItem** list for a tag.
GetTagData()	Obtain data corresponding to tag.
NextTagItem()	Iterate **TagItem** lists.
TagInArray()	Check if a tag value appears in a **Tag** array.
FilterTagChanges()	Eliminate **TagItems** which specify no change.
FilterTagItems()	Remove selected items from a **TagItem** list.
MapTags()	Convert **ti_Tag** values in a list via map pairing.
PackBoolTags()	Builds a "Flag" word from a **TagItem** list.

CALLBACK HOOK FUNCTION REFERENCE

The following are brief descriptions of the utility library functions which pertain to callback hooks.

Table 37-7: Utility Hook Functions

Function	Description
CallHookPkt()	Call a standard callback **Hook** function.

32-BIT INTEGER MATH FUNCTION REFERENCE

The following are brief descriptions of the utility library functions which pertain to 32-bit integer math.

Table 37-8: Utility 32-Bit Math Functions

Function	Description
SDivMod32()	Signed 32 by 32-bit division and modulus.
SMult32()	Signed 32 by 32-bit multiplication.
UDivMod32()	Unsigned 32 by 32-bit division modulus.
UMult32()	Unsigned 32 by 32-bit multiplication.

INTERNATIONAL STRING FUNCTION REFERENCE

The following are brief descriptions of the utility library functions which pertain to string operations using the international ASCII character set.

Table 37-9: Utility International String Functions

Function	Description
Stricmp()	Compare strings, case-insensitive.
Strnicmp()	Compare strings, case-insensitive, with specified length.
ToLower()	Convert a character to lower case.
ToUpper()	Convert a character to upper case.

DATE FUNCTION REFERENCE

The following are brief descriptions of the utility library functions which pertain to date conversion.

Table 37-10: Utility Date Functions

Function	Description
CheckDate()	Check the legality of a date.
Amiga2Date()	Calculate the date from a specified timestamp.
Date2Amiga()	Calculate the timestamp from a specified date.

Appendix A
LINKER LIBRARIES

This section describes the *amiga.lib* and *debug.lib* libraries. Unlike the libraries described in the other chapters of the manual, these are not shared run-time libraries. Code from the linker libraries is inserted by the linker into your final program. Only the functions you use are pulled into your code. These libraries are typically supplied by your language or compiler vendor.

AMIGA.LIB

This is the main Amiga scanned linker library, linked with most programs for the Amiga. The major components of *amiga.lib* are:

- *stubs* — Functions for each Amiga ROM routine that copy arguments from the stack to the CPU registers -- thereby enabling stack-based C compilers to call register-based Amiga ROM routines.
- *offsets* — The negative offset from the library base for each Amiga function. These are called *Library Vector Offsets* (_LVO).
- *Exec* — C functions which simplify many Exec procedures such as the creation and deletion of tasks, ports, and I/O request structures.
- *clib* — C support functions including pseudo-random number generation and a limited set of file and stdio functions designed to work directly with AmigaDOS file handles.
- *Math* — C functions which provide some basic conversions to and from Fast Floating Point (FFP) format numbers.
- *Graphics* — C support functions to add and remove tasks from the vertical-blanking interval interrupt server chain.
- *ARexx* — C support functions for ARexx variable handling and message checking.

NOTE: The Timer, Commodities, and Intuition support functions listed below are valid only for use with Release 2.04 (V37) or a later version of the system software.

- *Timer* C support functions to do common timer device operations.
- *Commodities* C functions which support the Commodities system. Included are functions to deal with ToolTypes, and to create various Commodities objects.
- *Intuition* Functions which provide support for Intuition's hook and Boopsi sub-systems.

DEBUG.LIB

This link library contains standard I/O (stdio) style functions for communicating with a serial terminal connected to the Amiga via its built-in serial port. Typically this terminal will be a 9600-baud, 8 data-bits, one stop-bit connection to an external terminal or an Amiga running a terminal package. The *debug.lib* functions allow you to output messages and prompt for input (even from within low level task or interrupt code) without disturbing the Amiga's display and or current state (other than the serial hardware itself).

No matter how badly the system may have crashed, these functions can usually get a message out. A similar debugging library (currently called *ddebug.lib)* is available for sending debugging output to the parallel port. This is useful for debugging serial applications. Ddebug.lib is not documented here. It contains functions similar to debug.lib but with names starting with 'D' instead of 'K'.

Please refer to the *Amiga ROM Kernel Reference Manual: Includes and Autodocs* for a detailed description of the functions.

Amiga.lib

Amiga.lib is the main linker library. Most applications link with and use at least one function in *amiga.lib.* The functions available are as follows.

EXEC SUPPORT

BeginIO()

This function takes an **IORequest** and passes it directly to the BEGINIO vector of the proper device. This works exactly like **SendIO()**, but does not clear the **io_Flags** field first. This function does not wait for the I/O to complete.

CreateExtIO() and **DeleteExtIO()**

CreateExtIO() allocates memory for and initializes a new I/O request block of a program-specified number of bytes. The number of bytes *must* be the size of a legal **IORequest** (or extended request) or very nasty things will happen. **DeleteExtIO()** frees up an I/O request as allocated by **CreateExtIO()**. The **mn_Length** field determines how much memory to deallocate.

CreatePort() and **DeletePort()**

CreatePort() allocates and initializes a new message port. The message list of the new port will be prepared for use via **NewList()**. The port will be set to signal your task when a message arrives (PA_SIGNAL). **DeletePort()** deletes the port created by **CreatePort()**. All messages that may have been attached to that port must already have been replied to.

CreateStdIO() and **DeleteStdIO()**

CreateStdIO() allocates memory for and initializes a new **IOStdReq** structure. **DeleteStdIO()** frees up an **IOStdReq** allocated by **CreateStdIO()**.

CreateTask() and **DeleteTask()**

These functions simplify creation and deletion of subtasks by dynamically allocating and initializing the required structures and stack space. They also add the task to Exec's task list with the given name and priority. A **tc_MemEntry** list is provided so that all stack and structure memory allocated by **CreateTask()** is automatically deallocated when the task is removed. Before deleting a task with **DeleteTask()**, you must first make sure that the task is not currently executing any system code which might try to signal the task after it is gone.

NewList()

Prepares a List structure for use; the list will be empty and ready to use.

CLIB

FastRand()

Generates a pseudo-random number. The seed value is taken from stack, shifted left one position, exclusive-or'ed with hex value $1D872B41 and returned.

RangeRand()

RangeRand() accepts a value from 1 to 65535, and returns a value within that range (a 16-bit integer). Note that this function is implemented in C.

fclose() Closes a file.

fgetc() Gets a character from a file.

fprintf() Prints a formatted output line to a file.

fputc() Puts character to file.

fputs() Writes a string to file.

getchar() Gets a character from *stdin*.

printf() Puts format data to *stdout*.

putchar() Puts character to *stdout*.

puts() Puts a string to *stdout*, followed by newline.

sprintf() Formats data into a string (see Exec **RawDoFmt()**).

MATH

afp() Converts ASCII string variable into fast floating-point.

arnd() ASCII round-off of the provided floating-point string.

dbf()

Accepts a dual-binary format floating-point number and converts it to FFP format.

fpa()

Accepts an FFP number and the address of the ASCII string where its converted output is to be stored. The number is converted to a NULL terminated ASCII string and stored at the address provided. Additionally, the base ten (10) exponent in binary form is returned.

GRAPHICS

AddTOF() and **RemTOF()**

AddTOF() adds a task to the vertical-blanking interval interrupt server chain. This frees C programmers from the burden of having to write an assembly language routine to perform this function. The task can be removed with **RemTOF()**.

AREXX

GetRexxVar() and **SetRexxVar()**

GetRexxVar() attempts to extract the value of a variable from a running ARexx script/program. **SetRexxVar()** will attempt to set the value of a particular variable in a running ARexx script.

CheckRexxMsg()

This function checks to make sure that a **RexxMsg** is from ARexx directly. Messages used by the Rexx Variable Interface (RVI) routines are required to be directly from ARexx.

TIMER

TimeDelay()

This function waits for a specified period of time before returning to the the caller.

COMMODITIES

ArgArrayInit() and **ArgArrayDone()**

ArgArrayInit() returns an array of strings suitable for sending to **icon.library/FindToolType().** This array will be the ToolTypes array of the program's icon, if it was started from Workbench. It will just be 'argv' if the program was started from a shell. **ArgArrayDone()** frees memory and does cleanup required after a call to **ArgArrayInit().**

ArgInt()

These functions look for a particular entry in a ToolType array returned by **ArgArrayInit()** and return the integer (**ArgInt()**) or string (**ArgString()**) associated with that entry. A default value can be passed to each function which will be returned in the event that the requested entry could not be found in the ToolType array.

CxCustom()

This function creates a custom commodity object. The action of this object on receiving a commodity message is to call a function of the application programmer's choice.

CxDebug()

This function creates a Commodities debug object. The action of this object on receiving a Commodities message is to print out information about the message through the serial port (using the **debug.lib/kprintf()** routine). A specified 'id' will also be displayed.

CxFilter()

Creates a Commodities input event filter object that matches a description string. The description string is in the same format as strings expected by **commodities.library/SetFilter().** If the description string is NULL, the filter will not match any messages.

CxSender()

This function creates a Commodities sender object. The action of this object on receiving a Commodities message is to copy the Commodities message into a standard Exec Message, to put a supplied id in the message as well, and to send the message off to the message port.

CxSignal()

This function creates a Commodities signal object. The action of this object on receiving a Commodities message is to send a signal to a task. The caller is responsible for allocating the signal and determining the proper task ID.

CxTranslate()

This function creates a Commodities translator object. The action of this object on receiving a Commodities message is to replace that message in the commodities network with a chain of Commodities input messages.

HotKey()

This function creates a triad of commodity objects to accomplish a high-level function.

The three objects are a filter, which is created to match by **CxFilter()**, a sender created by **CxSender()**, and a translator which is created by **CxTranslate()**, so that it swallows any commodity input event messages that are passed down by the filter.

This is the simple way to get a message sent to your program when the user performs a particular input action.

InvertString()

This function returns a linked list of input events which would translate into the string using the supplied keymap (or the system default keymap if the supplied keymap is NULL).

This chain should eventually be freed using **FreeIEvents().**

FreeIEvents()

This function frees a linked list of input events as obtained from **InvertString()**.

INTUITION

CallHook() and **CallHookA()**
: These functions invoke hooks. **CallHook()** expects a parameter packet ("message") on the stack, while **CallHookA()** takes a pointer to the message.

DoMethod() and **DoMethodA()**
: Boopsi support functions that ask a specified Boopsi object to perform a specific message. The message is passed in the function call for **DoMethodA()** and on the stack for **DoMethod()**. The message is invoked on the object's true class.

DoSuperMethod() and **DoSuperMethodA()**
: Boopsi support functions that ask a Boopsi object to perform a supplied message as if it was an instance of its superclass. The message is passed in the function call for **DoSuperMethodA()** and on the stack for **DoSuperMethod()**.

CoerceMethod() and **CoerceMethodA()**
: Boopsi support functions that ask a Boopsi object to perform a supplied message as if it was an instance of some other class. The message is passed in the function call for **CoerceMethodA()** and on the stack for **CoerceMethod.**

SetSuperAttrs()
: Boopsi support function which invokes the OM_SET method on the superclass of the supplied class for the supplied object. Allows the ops_AttrList to be supplied on the stack (i.e. in a varargs way).

Debug.lib

Debug.lib is a link library that provides useful diagnostic functions that are handy for developing code. It includes the following functions:

KCmpStr() Compare two null-terminated strings.

KGetChar() Get a character from the console.

KGetNum() Get a number from the console.

KMayGetChar() Return a character if present, but don't wait.

KPrintF() Print formatted data to the console.

KPutChar() Put a character to the console.

KPutStr() Put a string to the console.

Appendix B
BOOPSI CLASS REFERENCE

The *Boopsi Class Reference* documents all of the classes built into Intuition. Each class entry in the reference starts off with:

```
Class:          The class's name (for example, gadgetclass)
Superclass:     The class's superclass (for example, rootclass)
Include File:   The class's include file (for example, <intuition/gadgetclass.h>)
```

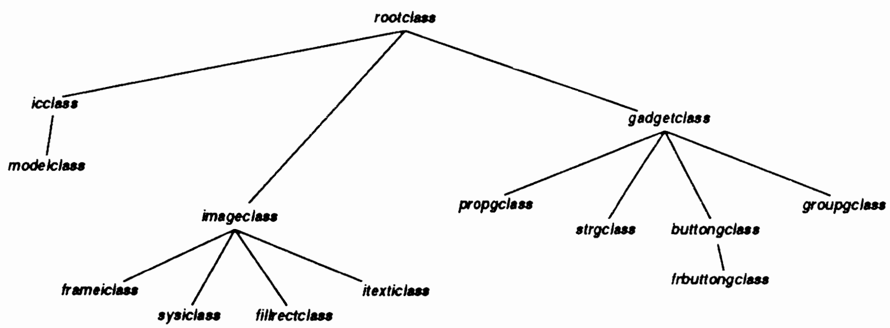
The include file contains the class's message structures, attribute IDs, and method IDs. This is followed by a short description of the class.

The rest of a class entry is broken up into three sections:

□ **New Methods**	Describes each new method that the class defines.
□ **Changed Methods**	Describes each method to which this class makes significant changes.
□ **Attributes**	Describes the attributes that this class defines as well as inherited ones that this class alters.

Appendix B: Boopsi Class Reference

There are 14 public classes built into the Release 2.04 ROM:



This appendix documents all the standard Boopsi classes, including their methods and attributes.

Each class entry in this document can have two sets of methods: new methods that the class defines and inherited methods that the class has modified significantly. Similarly, each class entry can have two sets of attributes: those that the class defines and those that the class inherited and modified. Unless documented otherwise, all classes inherit all of its superclass's methods and attributes.

Each method has a Boopsi message associated with it. These messages are in the form of C structures. Many methods use the default message structure:

```
typedef struct
{
    ULONG MethodID;
} *Msg;
```

Some methods require a customized message so they can pass other parameters besides the Method ID. If a method requires a custom message, its custom message structure is documented along with the method.

All methods have to return some sort of ***return value***. The meaning of the return value depends on the class and method. Normally a return value of zero indicates that either the method failed or it is not supported by the class. A method can use the return value to return a pointer to an object. If a class does not directly support a particular method, the class's dispatcher should pass the method on to its superclass for processing. The class dispatcher should record the return value it gets from its superclass and use that as its return value. Methods that assign no meaning to their return value can return 1L to indicate that the method is implemented.

The description of each attribute contains a code which lists the rootclass methods that apply to that attribute:

```
I    OM_NEW       Attribute can be set at initialization
S    OM_SET       Attribute can be set with OM_SET method
G    OM_GET       Attribute can be read with OM_GET method
N    OM_NOTIFY    Changing the attribute triggers object to send notifications
U    OM_UPDATE    Attribute can be set with OM_UPDATE method
```

For example, the itexticlass attribute `IA_Left` has the code (ISG) after it. This means an application can set `IA_Left` when it creates an instance of itexticlass (`OM_NEW`) and when it uses the `OM_SET` method. The application can also ask an itexticlass object what the `IA_Left` value is, using the `OM_GET` method.

The `OM_NEW`, `OM_SET`, `OM_NOTIFY`, and `OM_UPDATE` messages all contain a pointer to a tag list. This tag list contains the attributes and corresponding values that the method affects. Each TagItem in this list makes up an attribute/value pair. The ti_Tag portion of the TagItem contains the attribute's ID while the ti_Data field contains the attribute's value. Note that these tag lists can also contain *utility.library* Global System control tags (like `TAG_SKIP` and `TAG_DONE`), so dispatchers should use the tag functions from *utility.library* to process these lists. See documentation on the Utility library for more information.

All methods are called via a class dispatcher:

```
classDispatcher(Class *class, Object *object, Msg msg);
```

The first argument, class, is a pointer to the dispatcher's Class structure (defined in *<intuition/classes.h>*). The second parameter, object, is a pointer to the Boopsi object to which the Boopsi message (the third argument, msg) refers. Both Object and Msg are defined in *<intuition/classusr.h>*.

Class:	rootclass
Superclass:	None
Include File:	*<intuition/classusr.h>*

This is the universal base class for all other classes.

New Methods:

OM_NEW - This method tells a class to create a new instance of itself. If OM_NEW is successful, it returns a pointer to the new object, otherwise it returns NULL.

For programmers who are only creating Boopsi objects rather than creating custom classes, use the *intuition.library* function **NewObject()**:

```
APTR NewObject(struct IClass  *privateclass,
              UBYTE          *publicclassname,
              ULONG          firsttag,
              ...)
```

The OM_NEW method receives the following arguments (defined in *<intuition/classusr.h>*):

```
struct opSet /* The OM_NEW method uses the same structure as OM_GET */
{
    ULONG             MethodID;      /* OM_NEW */
    struct TagItem    *ops_AttrList; /* tag list of attributes to initialize */
    struct GadgetInfo *ops_GInfo;    /* Always NULL for OM_NEW */
};
```

The ops_AttrList field contains a pointer to a tag list of attribute/value pairs. Each pair contains an attribute ID and the initial value of the corresponding attribute.

The ops_GInfo field is always NULL for the OM_NEW method.

Unlike other methods, when the dispatcher gets an OM_NEW message, the object pointer (newobject from the **dispatchRKMModel()** prototype above) does not point to an object, since the idea is to create a new object. The pointer normally used to pass a Boopsi object is instead used to pass the address of the object's ''true class'' (the class of which the object is an instance).

The first thing the dispatcher does when it processes an OM_NEW message is pass the OM_NEW message on to its superclass's dispatcher. It does this using the *amiga.lib* function **DoSuperMethodA()**:

```
ULONG DoSuperMethodA(Class *cl, Object *trueclass, Msg msg);
```

Each superclass's dispatcher does this until the message gets to the rootclass dispatcher.

Each class keeps a record of how much memory its local instance data requires. The rootclass dispatcher's OM_NEW method looks at the object's true class (newobject from the prototype) to find out how much memory to allocate for the object's instance data. The rootclass dispatcher allocates enough memory for the true class's local instance data, plus enough memory for the local instance data of each of the true class's superclasses. If all goes well, the rootclass dispatcher increments the true class's internal count of instances of true class, and returns a pointer to the newly created object. it passes control back to the subclass dispatcher that called it. If there was a problem, the rootclass dispatcher passes back a NULL.

When the rootclass dispatcher returns, the subclass dispatcher regains control from **DoSuperMethodA()**. **DoSuperMethodA()** will return either a pointer to the new object or NULL if there was an error. Although the rootclass dispatcher allocated all the memory the object needs, it did not set up any of that memory. Now its the the subclass dispatcher's turn to do some work. It has to initialize the instance data that is local to its class. A dispatcher finds its local instance data by using the **INST_DATA()** macro (defined in *<intuition/classes.h>*).

After initializing its local instance data, the subclass dispatcher passes control down to its subclass dispatcher, which in turn, initializes its local instance data. Control passes down from class to subclass until the true class dispatcher regains control.

Now the true class dispatcher has to initialize its local instance data. It has to scan through the tag list of attribute/value pairs passed in the OM_NEW message (opSet.ops_AttrList). If the dispatcher finds any attributes that the true class recognizes, it has to initialize them to the value passed in the attribute/value pair.

At this point, the new object can allocate other resources it needs that it did not allocate as part of its instance data. For example, the new Boopsi object might need a frame image around itself, so it can create a new one using a Boopsi frame image. If the object allocates any resources in this step, it must deallocate these resources later when it is disposed in the OM_DISPOSE method.

Finally, the dispatcher can return. When the dispatcher returns from an OM_NEW method, it returns a pointer to the new object.

OM_DISPOSE - This method instructs an object to delete itself. The rootclass dispatcher's OM_DISPOSE method decrements the true class's internal count of instances of true class. The return value for this method is not explicitly defined.

This method uses the default Boopsi message.

Applications should not call this method directly. Instead they should use the *intuition.library* function **DisposeObject()**.

For the OM_DISPOSE method, an object should do the following:

Free any additional resources the object explicitly allocated itself in the OM_NEW method (this does not include the instance data).

Pass the message up to the superclass, which will eventually reach rootclass, which will free the instance data allocated for the object.

If a class does not allocate any extra resources when it creates an object, it can defer all OM_DISPOSE processing to its superclass.

OM_ADDTAIL - This method tells an object to add itself to the end of a specified Exec list. Boopsi objects contain a MinNode structure used for this purpose. The return value for this method is not explicitly defined.

The method uses a custom message (defined in <*intuition/classusr.h*>):

```
struct opAddTail {
    ULONG       MethodID;   /* OM_ADDTAIL */
    struct List *opat_List; /* The exec list to add the object to */
};
```

The opat_List can be any Exec list. Use the Intuition function **NextObject()** to step through this list.

OM_REMOVE - Remove an object from an Exec list. The return value for this method is not explicitly defined. This method uses the default Boopsi message.

The following methods are described at the rootclass level although its up to the subclasses to actually implement them. If a class does not implement these methods, it should either return zero, indicating that this class does not support the method, or defer processing on to its superclass.

OM_ADDMEMBER - Tells an object to add another object to its personal Exec list. What the list is for depends on the class. The return value for this method is not explicitly defined.

One class that uses this method is modelclass. A modelclass object maintains a broadcast list. When a modelclass object gets an OM_NOTIFY message, it broadcasts an OM_UPDATE message about the OM_NOTIFY to every object in its broadcast list.

This method uses a custom message (defined in <*intuition/classusr.h*>):

```
#define  opAddMember opMember
struct opMember {
    ULONG  MethodID;      /* OM_ADDMEMBER (or OM_REMMEMBER) */
    Object *opam_Object;  /* add (or remove) this object    */
};                        /* to (from) personal list.       */
```

opam_Object is the object to add to the list. A dispatcher typically implements OM_ADDMEMBER by sending the OM_ADDTAIL message to the opam_Object object.

OM_REMMEMBER - Tells an object to remove a member object from its personal list. The member object should have already been added with OM_ADDMEMBER. This method uses the same custom message as OM_ADDMEMBER. Normally a dispatcher implements OM_REMMEMBER by sending the OM_REMOVE message to the opam_Object object. The return value for this method is not explicitly defined.

OM_GET - Tells an object to report an attribute's value. Applications should not call this method directly. Instead, use the *intuition.library* function **GetAttr()**. The return value for this method is not explicitly defined.

This method uses a custom message (defined in <*intuition/classusr.h*>):

```
struct opGet {
    ULONG MethodID;      /* OM_GET */
    ULONG opg_AttrID;    /* ID of attribute to get */
    ULONG *opg_Storage;  /* place to put attribute value */
};
```

If the object's dispatcher recognizes opg_AttrID as one of the attributes defined by this class, the dispatcher should copy the value of that attribute to where opg_Storage points:

```
struct opGet *myopget;
...
*(myopget->opg_Storage) = my_attribute_value;
...
```

If the dispatcher does not recognize opg_AttrID, it should pass the message on to the superclass.

OM_SET - This method tells an object to set one or more of its attributes. Applications should not call this method directly. Instead, use the *intuition.library* functions **SetAttrs()** and **SetGadgetAttrs()** to call this method. The return value for this method is not explicitly defined.

The return value for this method is not explicitly defined. However, in general, when implementing the OM_SET method, if setting an object attribute causes some sort of visual state change, the OM_SET method should return a value greater than zero. If changing an attribute does not affect the visual state, OM_SET should return zero.

This method uses a custom message (defined in <*intuition/classusr.h*>):

```
struct opSet {
    ULONG              MethodID;      /* OM_SET */
    struct TagItem     *ops_AttrList; /* tag list of attributes to set */
    struct GadgetInfo  *ops_GInfo;
};
```

The ops_AttrList field contains a pointer to a tag list of attribute/value pairs. These pairs contain the IDs and the new values of the attributes to set. The dispatcher has to look through this list (see docs for the *utility.library* **NextTagItem()** function) for attributes its class recognizes and set the attribute's value accordingly. The dispatcher should let its superclass handle any attributes it does not recognize.

If the object is a gadget, the ops_GInfo field contains a pointer to a GadgetInfo structure. Otherwise, the value in ops_GInfo is undefined. Intuition use the GadgetInfo structure to pass display information to gadgets. See the gadgetclass methods for more details.

OM_UPDATE - This method tells an object to update one or more of its attributes. No application should call this method. Only Boopsi objects send OM_UPDATE messages. The return value for this method is not explicitly defined.

A Boopsi object uses an OM_UPDATE message to notify another Boopsi object about transitory changes to one or more attributes.

From the point of view of most objects, an OM_UPDATE message is almost identical to OM_SET. Because the methods are so similar, When a typical dispatcher receives an OM_UPDATE message, it processes the OM_UPDATE the same way it would process an OM_SET message, usually using the same code.

There are actually two kinds of OM_UPDATE, an interim and final one. While a Boopsi object's attribute is in a transient state, it can send out interim OM_UPDATE messages to its target(s). For example, while the user is sliding a Boopsi prop gadget, the prop gadget sends interim OM_UPDATE message about changes to its PGA_Top value (the integer value of the prop gadget is the PGA_Top attribute) to some target object. When the user lets go of the prop gadget, the gadget is no longer in a transient state, so the gadget sends out a final OM_UPDATE about its PGA_Top attribute. The target object can choose to change one of its attributes based on the OM_UPDATE messages it receives.

The layout of the OM_UPDATE message is almost identical to the OM_SET message:

```
struct opUpdate {  /* the OM_NOTIFY method also uses this structure */
    ULONG             MethodID;          /* OM_UPDATE              */
    struct TagItem    *opu_AttrList;     /* tag list of attributes */
    struct GadgetInfo *opu_GInfo;        /*   that changed.        */
    ULONG             opu_Flags;         /* The extra field        */
};

#define OPUF_INTERIM(1<<0)
```

Some dispatchers need to know the difference between an interim and final OM_UPDATE. A dispatcher can tell the difference between an interim and final OM_UPDATE message because the OM_UPDATE message has an extra field for flags. If the low order bit (the **OPUF_INTERIM** bit) is set, this is an interim OM_UPDATE message. The interim flag is useful to a class that wants to ignore any interim messages, processing only final attribute values.

OM_NOTIFY - This method tells an object to broadcast an attribute change to a set of target objects using OM_UPDATE messages. The return value for this method is not explicitly defined.

The OM_NOTIFY method uses the same message structure as OM_UPDATE.

Most dispatchers do not handle the OM_NOTIFY message directly. Normally they inherit this method from a superclass, so they pass the OM_NOTIFY message on to the superclass dispatcher.

Although most dispatchers don't have to process OM_NOTIFY messages, most *do* have to send them. Whenever an object receives an OM_SET or OM_UPDATE about one of its attributes, it may need to notify other objects of the change. For example, when a prop gadget's PGA_Top value changes, its target object(s) need to hear about it.

If an object needs to notify other objects about a change to one or more of its attributes, it sends *itself* an OM_NOTIFY message. The OM_NOTIFY message will eventually end up in the hands of a superclass that understands OM_NOTIFY and it will send OM_UPDATE messages to the target objects.

Changed Methods:

Not applicable.

Attributes:

None.

Class:	icclass (interconnection class)
Superclass:	rootclass
Include File:	*<intuition/icclass.h>*

Base class of simple OM_UPDATE forwarding objects. When an icclass object gets an OM_UPDATE message, it maps the attributes in the OM_UPDATE message according to its mapping list (its ICA_MAP attribute) and forwards the OM_UPDATE to it target (its ICA_TARGET attribute).

New Methods:

None.

Changed Methods:

OM_SET - This method sets its attributes and returns 0. Note that this is not the same behavior as OM_NOTIFY.

OM_UPDATE/OM_NOTIFY - These methods tell the object to notify its ICA_TARGET of attribute changes by sending the target an OM_UPDATE message. If the object has an ICA_MAP, it maps the attribute IDs it finds to new attribute IDs. See the rootclass descriptions of OM_NOTIFY and OM_UPDATE for more information. The return value for this method is not explicitly defined.

Attributes:

ICA_TARGET (IS) - This attribute stores the address of the icclass object's target object. Whenever the icclass object receives an OM_NOTIFY or OM_UPDATE message, it forwards that message to its target in the form of an OM_UPDATE message. If the icclass object has an attribute mapping list (see the ICA_MAP attribute below), it also maps the OM_NOTIFY/OM_UPDATE message's attribute IDs to new ones before forwarding the message.

If the value of ICA_TARGET is ICTARGET_IDCMP, the icclass object sends an IDCMP_IDCMPUPDATE IntuiMessage to its window instead of forwarding an OM_UPDATE message. See the rootclass description of OM_UPDATE for more information.

ICA_MAP (IS) - This attribute points to a tag list of attribute mappings which the icclass object uses to change the attribute IDs of an OM_UPDATE's attribute/value pairs. For example, if an icclass object had the following ICA_MAP:

```
struct TagItem map[] =
{
    {PGA_Top, STRINGA_LongVal},
    {MYATTR, MYNEWATTR},
    {TAG_END, }
};
```

before sending an OM_UPDATE to its ICA_TARGET, the icclass object scans through the OM_UPDATE

message's attribute/value pairs looking for the PGA_Top and MYATTR attributes. If it finds the PGA_Top attribute, it changes PGA_Top to STRINGA_LongVal. Likewise, if the icclass object finds the MYATTR attribute, it changes MYATTR to MYNEWATTR. The icclass object does not disturb the attribute's value.

ICSPECIAL_CODE (*) - This is a dummy attribute for the ICA_MAP. If any attribute maps to ICSPECIAL_CODE and ICA_TARGET is ICTARGET_IDCMP, then the value of the mapped attribute will be copied into the IntuiMessage.Code field of the IDCMP_IDCMPUPDATE message (just the lower sixteen bits of the attribute value will fit).

Class:	modelclass
Superclass:	icclass
Include File:	<*intuition/icclass.h*>

A class of OM_UPDATE forwarding objects that have multiple targets. In addition to the features the modelclass object inherits from icclass, when a modelclass object gets an OM_UPDATE message, it forwards that OM_UPDATE message to all of the objects in its broadcast list.

New Methods:

None.

Changed Methods:

OM_ADDMEMBER - This method tells a model to add an object to its broadcast list. When the object disposes of itself, it will also dispose of any objects remaining on its broadcast list. The return value for this method is not explicitly defined. See the rootclass description of OM_ADDMEMBER for more information.

OM_REMMEMBER - This method tells a model to remove an object from its broadcast list. The return value for this method is not explicitly defined. See the rootclass description of OM_REMMEMBER for more information.

OM_DISPOSE - This method tells a model to dispose of itself plus the objects remaining on its broadcast list. The return value for this method is not explicitly defined.

OM_NOTIFY/OM_UPDATE - This method tells an object to forward the message in the form of an OM_UPDATE message to all the objects in its broadcast list. The modelclass does not map the attributes in these OM_UPDATE messages. Because modelclass inherits behavior from icclass, if the model has an ICA_TARGET and ICA_MAP, it will also send a mapped OM_UPDATE message to its ICA_TARGET. The return values for these methods are not explicitly defined. See the rootclass and icclass descriptions of these methods for more information.

Attributes:

None.

Class: imageclass
Superclass: rootclass
Include File: <*intuition/imageclass.h*>

This class is the base class for Boopsi Images. These images are backwards compatible with the conventional Intuition Images. Every Boopsi image has an Intuition Image structure embedded in it so Intuition can access the Boopsi image as a conventional Image structure when necessary. Normally there are no direct instances of this class, only instances of subclasses of imageclass.

New Methods:

IM_DRAW - This method tells an image object to draw itself. Applications should not call this method directly, instead use the *intuition.library* function **DrawImageState()**. The return value for this method is not explicitly defined.

The IM_DRAW method uses a custom message structure:

```
struct impDraw
{
    ULONG              MethodID;    /* IM_DRAW                   */
    struct RastPort *imp_RPort;     /* RastPort to render into   */
    struct
    {                               /* X and Y offset relative to */
        WORD X;                     /* the image's IA_Left and   */
        WORD Y;                     /* IA_Top attributes         */
    } imp_Offset;
    ULONG              imp_State;   /* Visual state of image     */
    struct DrawInfo *imp_DrInfo;    /* DrawInfo describing rendering area */
};
```

The imp_State field contains the visual state to render the image. The visual states (defined in <*intuition/imageclass.h*>) are:

IDS_NORMAL	Render using normal imagery. This is the only kind of imagery available to non-Boopsi images.
IDS_SELECTED	Render using ''selected'' imagery. ''Selected'' refers to the state of a gadget's imagery when it is the selected gadget.
IDS_DISABLED	Render using ''disabled'' imagery. ''Disabled'' refers to the state of a gadget's imagery when it is disabled. Typically, a disabled image has a ghosting pattern on top of it.
IDS_INACTIVENORMAL	This is a special version of IDS_NORMAL for a ''normal'' image that is in the border of an inactive window.
IDS_INACTIVESELECTED	This is a special version of IDS_SELECTED for a ''selected'' image that is in the border of an inactive window.
IDS_INACTIVEDISABLED	This is a special version of IDS_DISABLED for a ''disabled'' image that is in the border of an inactive window.
IDS_BUSY	Render using ''busy'' imagery as if the object was the image of a gadget in a busy state. The busy gadget state is not yet supported by Intuition.
IDS_INDETERMINATE	Render using ''indeterminate'' imagery as if the object was the image of a gadget in an indeterminate state. The indeterminate gadget state is not yet supported by Intuition.

Most image objects do not have different visual states for each possible imp_State. See the image class entries in this index for more details.

When setting the pens to render an image, use the values from the imp_DrInfo->dri_Pens pen array (Note that it is possible for imp_DrInfo to be NULL). The possible pen values are defined in <*intuition/screens.h*>. See the ''Intuition Screens'' chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information on the pen array.

IM_HITTEST - This method returns TRUE if a point is within the old Image structure box defined by the Image structure's LeftEdge, TopEdge, Width, and Height fields. Subclasses of imageclass can redefine this method if they need to change the criteria for deciding if a point is within an image. Application programs should not call this method directly, instead use the Intuition function **PointInImage()**. The IM_HITTEST method uses a custom message structure:

```
struct impHitTest
{
    ULONG MethodID; /* IM_HITTEST */
    struct
    {
        WORD X;     /* Coordinates of point to test for hit */
        WORD Y;
    } imp_Point;
};
```

If an image object doesn't need to make any changes to how its superclass handles IM_HITTEST, it can blindly pass this method on to its superclass.

IM_ERASE - The IM_ERASE method tells an image to erase itself. Applications should not call this method directly, instead they should call the Intuition function **EraseImage()**. The return value for this method is not explicitly defined.

The IM_ERASE method uses a custom message structure:

```
struct impErase
{
    ULONG              MethodID;     /* IM_ERASE */
    struct RastPort *imp_RPort;      /* The image's RastPort */
    struct
    {
        WORD X;                      /* X and Y offset relative */
        WORD Y;                      /* to the image's IA_Left  */
    } imp_Offset;                    /* and IA_Top attributes.  */
};
```

The imageclass dispatcher calls the *graphics.library* function **EraseRect()** to erase the image. The imageclass dispatcher gets the position of the image using the offsets from the IM_ERASE message and the dimensions it finds in the object's Image structure. The imageclass dispatcher does not do any bounds checking before calling **EraseRect()**.

IM_DRAWFRAME - The IM_DRAWFRAME method instructs an image object to render itself within the confines of a given rectangle. The return value for this method is not explicitly defined.

This method uses a custom message structure that is basically an extension of the IM_DRAW message:

```
struct impDraw
{
    ULONG            MethodID;     /* IM_DRAWFRAME */
    struct RastPort *imp_RPort;    /* RastPort to render into */
    struct
    {
        WORD X;                    /* X and Y offset relative to the */
        WORD Y;                    /* image's IA_Left and IA_Top attributes */
    } imp_Offset;
    ULONG            imp_State;    /* Visual state of image (see defines below */
    struct DrawInfo *imp_DrInfo;   /* DrawInfo describing target RastPort (can be NULL) */

    struct
    {
        WORD Width;                /* scale, clip, restrict, etc. to these bounds */
        WORD Height;
    } imp_Dimensions;
};
```

The Width and Height fields provide the object's rectangular bounds. How the image object deals with the frame is implementation specific. If the imageclass dispatcher sees this message, it will convert it to an IM_DRAW message and send it back to the image's true class. An image subclass which assigns proper meaning to this method (i.e., frameiclass) should handle this method itself.

This method is useful to classes of images that can scale or clip themselves to arbitrary dimensions. Typically, an instance of a class that truly supports this method will massage its imagery as best it can to fit into the rectangle.

In general, applications that use this method to draw an object should use the IM_ERASEFRAME method to erase it (see below). This will ensure that the image erases itself at the proper scale.

IM_HITFRAME - This method is a special version of IM_HITTEST for images that support IM_DRAWFRAME. It asks an image if a point would be inside it if the image was confined (scaled, clipped, etc.) to a rectangular bounds. The return value for this method is not explicitly defined.

This method uses a custom message structure:

```
struct impHitTest
{
    ULONG MethodID;     /* IM_HITFRAME */
    struct
    {
        WORD X;         /* Coordinates of point to test for hit */
        WORD Y;
    } imp_Point;

    struct
    {
        WORD Width;     /* scale, clip, restrict, etc. to these bounds */
        WORD Height;
    } imp_Dimensions;
};
```

The imageclass dispatcher treats IM_HITFRAME just like IM_HITTEST, ignoring the restricting dimensions.

IM_ERASEFRAME - This method is a special version of IM_ERASE for images that support IM_DRAWFRAME. It asks an image to erase itself as if it were confined (scaled, clipped, etc.) to a rectangular bounds. The return value for this method is not explicitly defined.

This method uses a custom message structure:

```
struct impErase /* NOTE: This is a subset of impDraw */
{
    ULONG            MethodID;    /* IM_ERASEFRAME */
    struct RastPort *imp_RPort;   /* The image's RastPort */
    struct
    {
        WORD X;                   /* X and Y offset relative to the */
        WORD Y;                   /* image's IA_Left and IA_Top attributes */
    } imp_Offset;

    struct
    {
        WORD Width;               /* scale, clip, restrict, etc. to these bounds */
        WORD Height;
    } imp_Dimensions;
};
```

The imageclass dispatcher handles an IM_ERASEFRAME message as if it was an IM_ERASE message, ignoring the bounds. See the imageclass description for IM_ERASE for more details.

IM_FRAMEBOX - This method applies to image classes that are used to put a frame centered around some other objects. This method asks a framing image what its dimensions should be if it had to frame some object or set of objects that fit into a rectangular bounds. For example, to draw an frameiclass image around a group of gadgets that fit into a specific rectangle, you first send the frameiclass object an IM_FRAMEBOX message describing the dimensions and position of that rectangle. The frame reports what its position and dimensions would have to be to surround those gadgets. Use these results to draw the frameiclass image. The return value for this method is not explicitly defined.

IM_FRAMEBOX uses a custom message structure:

```
struct impFrameBox
{
    ULONG            MethodID;          /* IM_FRAMEBOX  */
    struct IBox     *imp_ContentsBox;   /* The object fills in this structure */
                                        /* with the dimensions of a rectangle */
                                        /* big enough to frame...             */
    struct IBox     *imp_FrameBox;      /* <------- this rectangle.           */
    struct DrawInfo *imp_DrInfo;        /* imp_DrInfo can be NULL.            */
    ULONG            imp_FrameFlags;
};
#define FRAMEF_SPECIFY  (1<<0)  /* Make do with the dimensions passed  */
                                /* in FrameBox.                        */
```

The imp_FrameBox field points to an IBox structure (defined in <*intuition/intuition.h*>) describing the dimensions and position of the rectangle to frame. After the framing image determines the position and size it should be in order to properly frame imp_FrameBox, it stores the result in the imp_ContentsBox IBox. This method allows an application to use a framing image without having to worry about image specific details such as the thickness of the frame or centering the frame around the object.

The imp_FrameFlags field is a bit field used to specify certain options for the IM_FRAMEBOX method. Currently, there is only one defined for it, FRAMEF_SPECIFY. If this bit is set, the imp_FrameBox contains a width and height that the frame image has to use as its width and

height, even if the imp_FrameBox is smaller than imp_ContentsBox. The frame is free to adjust its position, but it is stuck with the imp_FrameBox dimensions. This allows an application to set the size of the frame image and still allow the frame image to position itself so it is centered on a rectangle.

The imageclass dispatcher does not support this method. It returns zero.

Changed Methods:

OM_NEW - The instance data for imageclass contains an Image structure, and its Depth field is initialized to CUSTOMIMAGEDEPTH, which identifies such images to Intuition. The Image's Width and Height fields default to arbitrary positive numbers for safety, but an imageclass subclass or an application should set these attributes to something meaningful.

OM_SET - This method applies to all imageclass attributes. OM_SET returns 1.

Attributes:

IA_Left, IA_Top, IA_Width, IA_Height (ISG) - These attributes correspond to similarly named fields in the Intuition Image structure. The imageclass dispatcher stores these attributes in their corresponding fields in the image object's embedded Image structure.

IA_FGPen, IA_BGPen (ISG) - These attributes are copied into the Image structure's PlanePick and PlaneOnOff fields, respectively.

IA_Data (ISG) - A pointer to general image data. This value is stored in the ImageData field of the Image structure.

IA_Pens () - This attribute points to an alternative pen array for the image. Imageclass does not support this attribute, it is described here for subclasses to use. See the "Intuition Screens" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information on the pen array.

Class:	frameiclass
Superclass:	imageclass
Include File:	*<intuition/imageclass.h>*

This is a class of framing image, which can optionally fill itself. Its purpose is to frame other display elements using an embossed or recessed rectangular frame. The frame renders itself using the appropriate DrawInfo pens (SHINEPEN, SHADOWPEN, etc.). This class is intelligent enough to bound or center its contents.

New Methods:

None.

Changed Methods:

IM_DRAW - This method tells a frameiclass object to render itself using the position and dimensions of its Image structure. It supports two sets of drawing states (passes in the impDraw.imp_State field):

> IDS_NORMAL, IDS_INACTIVENORMAL, IDS_DISABLED- In these states, the frame renders its edges using SHADOWPEN and SHINEPEN. If it is a filled frame the frame uses the BACKGROUNDPEN for its interior. Note that the frame renders the same imagery for all three of these states.
>
> IDS_SELECTED, IDS_INACTIVESELECTED- In these states, the frame renders its edges using SHADOWPEN and SHINEPEN. If it is a filled frame the frame uses the FILLPEN for its interior.

See the imageclass description for IM_DRAW for more details.

IM_DRAWFRAME - This method is almost the same as the frameiclass IM_DRAW method, except this method accepts a width and height that overrides the width and height stored in the object's Image structure. It uses the same drawing states as the frameiclass's IM_DRAW method. See the imageclass description for IM_DRAWFRAME for more information.

IM_FRAMEBOX - This method asks a frameiclass image what its dimensions would be if it has to frame a specific rectangular area. See the imageclass description for IM_FRAMEBOX for more information.

Attributes:

IA_Recessed (IS) - If this attribute is TRUE, a frameiclass object will appear recessed into the drawing surface. It does this by swapping its use of the SHADOWPEN and SHINEPEN. By default, the frame appears to be raised from the surface.

IA_EdgesOnly (IS) - If this attribute is TRUE, the frame does not fill itself, it just draws its edges.

Class:	sysiclass
Superclass:	imageclass
Include File:	*<intuition/imageclass.h>*

This is a class of system images and standard application images. As of Intuition version 37, there are 11 possible sysiclass image glyphs to choose from:

DEPTHIMAGE	Window depth arrangement image.
ZOOMIMAGE	Window Zoom image.
SIZEIMAGE	Window Sizing image.
CLOSEIMAGE	Window close image.
SDEPTHIMAGE	Screen depth arrangement image.
LEFTIMAGE	Left arrow image.
RIGHTIMAGE	Right arrow image.
UPIMAGE	Up arrow image.
DOWNIMAGE	Down arrow image.
CHECKIMAGE	Checkmark image.
MXIMAGE	Radio button image.

The class caches the image's bitmap to improve rendering speed.

New Methods:

None.

Changed Methods:

None.

Attributes:

SYSIA_DrawInfo (I) - This attribute contains a pointer to a DrawInfo structure (defined in *<intuition/screens.h>*) describing the target screen. The class requires this attribute in order to generate the image into a bitmap cache.

SYSIA_Which (I) - This attribute identifies which of the system image glyphs the sysiclass object uses. It can be one of the 11 glyphs described above.

SYSIA_Size (I) - This attribute identifies which image size to use for the object. This generalizes Intuition's older concept of two different system image dimensions. There are three possible values for this attribute:

SYSISIZE_MEDRES	Meant for Hires, non-interlaced 640x200/256 display.
SYSISIZE_HIRES	Meant for Hires, interlaced 640x400/512 display.
SYSISIZE_LOWRES	Meant for Lores 320x200/256 display.

These sizes do not apply to all of the glyphs consistently. See the chart below for image dimensions (width x height) according to the SYSIA_Size and the glyph type. An 'H' for the height means the glyph allows its height to be specified with IA_Height.

SYSISIZE	LOWRES	SYSISIZE_MEDRES	SYSISIZE_HIRES
DEPTHIMAGE	18 x H	24 x H	24 x H
ZOOMIMAGE	18 x H	24 x H	24 x H
SIZEIMAGE	13 x 11	18 x 10	18 x 10
CLOSEIMAGE	15 x H	20 x H	20 x H
SDEPTHIMAGE	17 x H	23 x H	23 x H
LEFTIMAGE	16 x 11	16 x 10	23 x 22
RIGHTIMAGE	16 x 11	16 x 10	23 x 22
UPIMAGE	13 x 11	18 x 11	23 x 22
DOWNIMAGE	13 x 11	18 x 11	23 x 22
CHECKIMAGE	26 x 11	26 x 11	26 x 11
MXIMAGE	17 x 9	17 x 9	17 x 9

Class: fillrectclass
Superclass: imageclass
Include File: *<intuition/imageclass.h>*

This is a class of filled rectangles. The fillrectclass object can use a pattern to fill in its interior.

New Methods:

None.

Changed Methods:

IM_DRAW - This method asks a fillrectclass object to render itself relative to the position (LeftEdge and TopEdge) and dimensions (Width and Height) in its embedded Image structure. See the imageclass description of `IM_DRAW` for more details.

IM_DRAWFRAME - This method asks a fillrectclass object to render itself relative to the position in its embedded Image structure, but using the width and height passed in the message's Dimensions.Width and Dimensions.Height fields. See the imageclass description of `IM_DRAWFRAME` for more details.

Attributes:

IA_APattern, IA_APatSize (IS) - These attributes supply the fillrectclass object with an area fill pattern. The `IA_APattern` attribute points to the area fill pattern for the object. The `IA_APatSize` attribute is the depth of the area fill pattern. These attribute values are similar to the parameters passed to the **SetAfPt()** macro (defined in *<graphics/gfxmacros.h>*) and indirectly correspond to fields in a RastPort structure. For more information on these patterns, see the section on patterns in the "Graphics Primitives" chapter of the *Amiga ROM Kernel Reference Manual: Libraries.*

IA_Mode (IS) - This attribute contains the drawing mode for the pattern (JAM1, JAM2, etc.)

Class: itexticlass
Superclass: imageclass
Include File: *<intuition/imageclass.h>*

This is a class of image objects that render an IntuiText structure. Using some of the imageclass attributes, the object can override some of the parameters in the IntuiText structure. This class makes it easy to share an IntuiText structure between objects.

New Methods:

IM_DRAW/IM_DRAWFRAME - These methods ask an itexticlass object to render its IntuiText structure, which it gets from the imageclass `IA_Data` attribute. An itexticlass object renders its IntuiText relative to the `IA_Left` and `IA_Top` attributes it inherits from imageclass. This method uses the JAM1 drawing mode and the `IA_FGPen` to render the text. See the imageclass description of `IM_DRAW/IM_DRAWFRAME` for more details.

Changed Methods:

None.

Class: gadgetclass
Superclass: rootclass
Include File: <*intuition/gadgetclass.h*>

This is a base class for Intuition compatible gadget objects. The dispatcher for this class takes care of creating an Intuition Gadget structure as part of its local instance data. All of the standard Boopsi gadget classes build on this class. Normally there are no direct instances of this class, only instances of subclasses of gadgetclass.

The behavior of a Boopsi gadget depends on how it handles the five Boopsi gadget methods: GM_HITTEST, GM_RENDER, GM_GOACTIVE, GM_HANDLEINPUT, and GM_GOINACTIVE. Intuition controls a Boopsi gadget by sending it these types of messages. The structures that these methods use for their messages begin with the method's ID followed by a pointer to a GadgetInfo structure (defined in <*intuition/cghooks.h*>). The GadgetInfo structure is a read-only structure that contains information about the gadget's rendering environment. The gadget uses this to find things like its window, screen, or pen array. Although this structure does contain a pointer to a RastPort for the gadget, the gadget must not use this RastPort for rendering. The gadget can obtain a RastPort for rendering by calling the Intuition function **ObtainGIRPort()** using the GadgetInfo structure. See the *intuition.library* Autodocs for more details on this function.

These methods are not defined directly by gadgetclass. It is up to subclasses of gadgetclass to implement them.

Like all Boopsi methods, these methods run on the context of the task that called the method. Normally, Intuition is the only entity that calls these methods, so these normally operate in the *input.device's* task. Because a gadget may have to process a large number of input events, poor implementations of gadget methods (especially the GM_HANDLEINPUT method) can degrade system performance.

New Methods:

GM_HITTEST - This method asks a gadget if a point is within its bounds. Usually the point corresponds to a mouse click. Intuition sends a gadget this message when the user clicks inside the rectangular bounds found in the object's Gadget structure (using its TopEdge, LeftEdge, Width, and Height fields). This method returns GMR_GADGETHIT if a point is within the gadget, otherwise it returns zero. Because the gadget decides if it was hit, the gadget can be almost any shape or pattern. Boopsi gadgets that default to using the bounds of their Gadget structure should always return GMR_GADGETHIT.

GM_HITTEST uses a custom message structure (defined in <*intuition/gadgdetclass.h*>):

```
struct gpHitTest
{
    ULONG              MethodID;      /* GM_HITTEST    */
    struct GadgetInfo *gpht_GInfo;
    struct
    {
        WORD X;                       /* Is this point inside */
        WORD Y;                       /* of the gadget?       */
    } gpht_Mouse;
};
```

The gpht_Mouse.X and gpht_Mouse.Y fields make up the X and Y coordinates of the hit point. These coordinates are relative to the upper-left corner of the gadget (Gadget.LeftEdge, Gadget.TopEdge).

GM_RENDER - This method tells a gadget to render itself. The return value for this method is not explicitly defined.

GM_RENDER uses a custom message structure (defined in <*intuition/gadgetclass.h*>):

```
struct gpRender
{
    ULONG              MethodID;    /* GM_RENDER */
    struct GadgetInfo *gpr_GInfo;
    struct RastPort   *gpr_RPort;   /* all ready for use */
    LONG               gpr_Redraw;  /* might be a "highlight pass" */
};
```

The GM_RENDER message contains a pointer to the Gadget's RastPort which it can use for rendering. The Gadget renders itself according to how much imagery it needs to replace. The gpr_Redraw field contains one of three values:

GREDRAW_REDRAW	Redraw the entire gadget.
GREDRAW_UPDATE	The user has manipulated the gadget changing the imagery. Update only that part of the gadget's imagery that is effected by the user manipulating the gadget (for example, the knob and scrolling field of the prop gadget).
GREDRAW_TOGGLE	If this gadget supports it, toggle to or from the highlighting imagery.

GM_GOACTIVE - This method asks a gadget if it is OK to make it the active gadget. The active gadget is the gadget that is currently receiving user input. Intuition sends this message after a gadget responds affirmatively to the GM_HITTEST method. A gadget becomes active because it needs to process input events (like a prop gadget or string gadget).

Some types of gadget do not need to become active. These gadgets do not have to process input from the user, they only have to deal with a single mouse click to toggle their state. Because that mouse click triggered this method, the button already has all of the user input it requires. Note that the behavior of the GadTools button differs from a Boopsi buttongclass gadget, which processes other input events besides a single mouse click. See the entry for buttongclass in this Appendix for more details.

GM_GOACTIVE uses a custom message structure (defined in <*intuition/gadgetclass.h*>):

```
struct gpInput
{
    ULONG              MethodID;         /* GM_GOACTIVE or GM_HANDLEINPUT */
    struct GadgetInfo *gpi_GInfo;
    struct InputEvent *gpi_IEvent;       /* The input event that triggered this  */
                                         /* method (can be NULL for GM_GOACTIVE). */
    LONG              *gpi_Termination;  /* For GADGETUP IntuiMessage.Code       */
    struct
    {
        WORD X;                          /* Mouse position relative to upper     */
        WORD Y;                          /* left corner of gadget (LeftEdge,     */
    } gpi_Mouse;                         /* TopEdge).                            */
};
```

The gpi_IEvent field points to the struct InputEvent that triggered the GM_GOACTIVE message. If gpi_IEvent is NULL, the GM_GOACTIVE message was triggered by a function like *intuition.library's* **ActivateGadget()** and not by the user clicking the gadget.

For gadgets that only want to become active as a direct result of a mouse click, this difference is important. For example, the prop gadget becomes active only when the user clicks on its knob. Because the only way the user can control the prop gadget is via the mouse, it would not make sense for it to be activated by anything besides the mouse. On the other hand, a string gadget gets input from the keyboard, so a string gadget doesn't care what activates it. Its input comes from the keyboard rather than the mouse.

A gadget's GM_GOACTIVE method returns GMR_MEACTIVE (defined in <*intuition/gadgetclass.h*>) if it wants to be the active gadget. Otherwise it returns GMR_NOREUSE. For a description of what these values mean, See their description in the gadgetclass's GM_HANDLEINPUT method, below.

If necessary, a gadget's GM_GOACTIVE method can precalculate and cache information before it becomes the active gadget. The gadget will use this information while it's processing user input with the GM_HANDLEINPUT method. When it is time for the active gadget to become inactive, Intuition will send the gadget a GM_GOINACTIVE message. The gadget can clean up its precalculations and cache in the GM_GOINACTIVE method. For more information on GM_GOINACTIVE, see its description below.

GM_HANDLEINPUT - This method asks an active gadget to handle an input event. After Intuition gets an OK to make this gadget object active (see the GM_GOACTIVE method above), Intuition starts sending input events to the gadget. Intuition sends them in the form of a GM_HANDLEINPUT message. This method uses the same custom message structure as GM_GOACTIVE (see the gpInput structure above).

The information in the gpInput structure is the same for GM_HANDLEINPUT as it is for GM_GOACTIVE. The only difference is that the GM_HANDLEINPUT message's gpi_IEvent can never be NULL. It always points to an InputEvent structure.

The gadget has to examine the incoming InputEvents to see how its state may have changed. For example, a string gadget processes key presses, inserting them into the gadgets string. When the string changes, the gadget has to update its visual state to reflect that change. Another example is the prop gadget. If the user picks up the prop gadget's knob, the prop gadget has to track the mouse to process changes to the gadget's internal values. It does this by processing IECLASS_RAWMOUSE events.

If the GM_HANDLEINPUT method needs to do some rendering, it must call **ObtainGIRPort()** on the GM_HANDLEINPUT message's gpi_GInfo to get a pointer to a RastPort. To relinquish this RastPort, the GM_HANDLEINPUT method must call **ReleaseGIRPort()**. The GM_HANDLEINPUT method has to allocate and release this RastPort, it cannot be cached in the GM_GOACTIVE method.

The return value from GM_HANDLEINPUT informs Intuition if the gadget wants to remain active. The return values for the GM_HANDLEINPUT are similar to GM_GOACTIVE. The gadget tells Intuition that it wants to remain active by returning GMR_MEACTIVE. A gadget tells Intuition it wants to become inactive by returning one of the "go inactive" return values:

GMR_NOREUSE	Tells Intuition to throw away the gpInput.gpi_IEvent InputEvent.
GMR_REUSE	Tells Intuition to reprocess the gpInput.gpi_IEvent InputEvent after deactivating the gadget.
GMR_NEXTACTIVE	Tells Intuition to throw away the gpInput.gpi_IEvent InputEvent and activate the next GFLG_TagCycle gadget.
GMR_PREVACTIVE	Tells Intuition to throw away the gpInput.gpi_IEvent InputEvent and activate the previous GFLG_TagCycle gadget.

GMR_NOREUSE tells Intuition that the gadget does not want to be active and should throw away the InputEvent that triggered the GM_HANDLEINPUT message (or the GM_GOACTIVE message). For example, an active prop gadget returns GMR_NOREUSE when the user lets go of the left mouse button (thus letting go of the prop gadget's knob).

A gadget can also return GMR_REUSE, which tells Intuition to reuse the InputEvent. For example, if the user clicks outside of an active string gadget, that string gadget returns GMR_REUSE so Intuition can process that mouse click, which could be over another gadget. Another case where a string gadget returns GMR_REUSE is when the user pushes the right mouse button (the menu button). The string gadget becomes inactive and the menu button InputEvent gets reused by Intuition so it can pop up the menu bar.

The other two possible return values, GMR_NEXTACTIVE and GMR_PREVACTIVE were added to the OS for Release 2.04. These tell Intuition that a gadget no longer wants to be active and that the GM_HANDLEINPUT message InputEvent should be discarded. Intuition then looks for the next non-disabled (GMR_NEXTACTIVE) or previous (GMR_PREVACTIVE) gadget that has its GFLG_TABCYCLE flag set in its Gadget.Activation field (see the gadgetclass GA_TabCycle attribute below), and attempts to activate it.

For both GM_GOACTIVE and GM_HANDLEINPUT, the gadget can bitwise OR any of these "go inactive" return values with GMR_VERIFY. The GMR_VERIFY flag tells Intuition to send a GADGETUP IntuiMessage to the gadget's window. If the gadget uses GMR_VERIFY, it has to supply a value for the IntuiMessage's Code field. It does this by passing a value in the gpInput's gpi_Termination field. This field points to a long word, the lower 16-bits of which Intuition copies into the Code field. The upper 16-bits are for future enhancements, so clear these bits.

GM_GOINACTIVE - This method tells the active gadget to become inactive. The return value for this method is not explicitly defined.

GM_GOINACTIVE uses a custom message structure (defined in <*intuition/gadgetclass.h*>):

```
struct gpGoInactive
{
    ULONG             MethodID;    /* GM_GOINACTIVE */
    struct GadgetInfo *gpgi_GInfo;

/* V37 field only!  DO NOT attempt to read under V36! */
    ULONG             gpgi_Abort; /* gpgi_Abort=1 if gadget was aborted */
                                  /* by Intuition and 0 if gadget went  */
                                  /* inactive at its own request.       */
};
```

The gpgi_Abort field contains either a 0 or 1. If it is 0, the gadget became inactive at its own request (because the GM_HANDLEINPUT method returned something besides GMR_MEACTIVE). If gpgi_Abort is 1, Intuition aborted this active gadget. Some cases where Intuition aborts a gadget include: the user clicked in another window or screen, an application removed the active gadget with **RemoveGList()**, and an application called **ActivateWindow()** on a window other than the gadget's window.

If the gadget allocated any resources to cache or precalculate information in the GM_GOACTIVE method, it should deallocate those resources in this method.

Changed Methods:

OM_NEW - This method allocates space for an embedded struct Gadget (defined in <*intuition/intuition.h*>) and initializes some of the attributes defined by gadgetclass.

OM_NOTIFY - This method tells a gadget to send an OM_UPDATE message to its target object. Boopsi gadgets have a function similar to icclass objects--each gadget can have an ICA_TARGET and ICA_MAP in order to notify some target object of attribute changes. When a Boopsi gadget sends an OM_NOTIFY message, it always includes its GA_ID. This makes it easy for an application to tell which gadget initially sent the OM_NOTIFY. See the description of icclass and the rootclass's OM_NOTIFY and OM_UPDATE methods for more details.

Attributes:

GA_Previous (I) - This attribute is used to insert a new gadget into a list of gadgets linked by their Gadget.NextGadget field. When the OM_NEW method creates the new gadget, it inserts the new gadget into the list following the GA_Previous gadget. This attribute is a pointer to the gadget (struct Gadget *) that the new gadget will follow. This attribute cannot be used to link new gadgets into the gadget list of an open window or requester, use **AddGList()** instead.

ICA_TARGET (IS) - This attribute stores the address of the gadget's target object. Whenever the gadget receives an OM_NOTIFY message, it sends an OM_UPDATE message to its target. If the gadget has an attribute mapping list (see the ICA_MAP attribute below), it also maps the IDs from the OM_NOTIFY message.

If the value of ICA_TARGET is ICTARGET_IDCMP, the gadget sends an IDCMP_IDCMPUPDATE IntuiMessage to its window. See the rootclass description of OM_UPDATE for more information.

ICA_MAP (IS) - This attribute points to a tag list of attribute mappings which the gadget uses to change the attribute IDs of an OM_UPDATE's attribute/value pairs. For example, if a gadget had the following ICA_MAP:

```
struct TagItem map[] =
{
    {PGA_Top, STRINGA_LongVal},
    {MYATTR, MYNEWATTR},
    {TAG_END, }
};
```

before it sends an OM_UPDATE to its ICA_TARGET, the gadget scans through the OM_UPDATE message's attribute/value pairs looking for the PGA_Top and MYATTR attributes. If it finds the PGA_Top attribute, it changes PGA_Top to STRINGA_LongVal. Likewise, if the gadget finds the MYATTR attribute, it changes MYATTR to MYNEWATTR. The gadget does not disturb the attribute's value, only its ID.

GA_Left, GA_Top, GA_Width, GA_Height (IS) - These attributes correspond to the Gadget structure's LeftEdge, TopEdge, Width, and Height fields. Setting these clears the "gadget relative" flags (below).

GA_RelRight, GA_RelBottom, GA_RelWidth, GA_RelHeight (IS) - These attributes correspond to the Gadget structure's LeftEdge, TopEdge, Width, and Height fields. Setting any of these attributes also sets the corresponding "relative" flag in the Gadget structure's Flags field (respectively, GFLG_RELRIGHT, GFLG_RELBOTTOM, GFLG_RELWIDTH, and GFLG_RELHEIGHT). Note that the value passed in this attribute is normally a negative LONG. See the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information on these flags.

The remaining attributes defined by gadgetclass are used to set the fields in the Gadget structure of the Boopsi gadget. Some Boopsi gadgets do not pay attention to many of the fields in its Gadget structure, so most applications will not have to worry about the majority of these attributes. Some gadget classes assign special meanings to these attributes. See the documentation of the specific gadget classes for more details.

GA_IntuiText, GA_Text, GA_LabelImage (IS) - These attributes correspond to one field in the object's embedded Gadget structure--the GadgetText field. Setting any of these attributes copies the attribute's value blindly into the GadgetText field. In addition, setting GA_Text also sets the GFLG_LABELSTRING flag in Gadget.Flags and setting GA_LabelImage sets the GFLG_LABELIMAGE flag in Gadget.Flags. The GA_IntuiText attribute must be an IntuiText pointer, as with old-style gadgets. GA_Text takes a pointer to a NULL- terminated string (UBYTE *). GA_LabelImage takes a pointer to a (Boopsi) image. Note that most gadget classes do not support GA_Text and GA_LabelImage. See the description of specific gadget classes for more details.

GA_Image (IS) - This attibute is a pointer to either a Boopsi image or a Release 1.3-compatible Intuition image. This attribute corresponds to the Gadget's GadgetRender field. The gadgetclass dispatcher will not dispose of this image when it disposes of the gadget object.

GA_Border, GA_SelectRender, GA_ID, GA_UserData, GA_SpecialInfo (IS) - These attributes correspond to the similarly named fields in the Gadget structure embedded in the gadget object.

GA_GZZGadget, GA_SysGadget (IS) - These are boolean attributes that correspond to the flags in the object's Gadget.GadgetType field. If the value passed with the attribute is TRUE, the corresponding flag in Gadget.GadgetType is set. If the value passed with the attribute is FALSE, the corresponding flag in Gadget.GadgetType is cleared. See the *<intuition/intuition.h>* include file or the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

GA_Disabled, GA_Selected (IS) - These are boolean attributes that correspond to the similarly named flags in the object's Gadget.Flags field. If the value passed with the attribute is TRUE, the corresponding flag in Gadget.Flags is set. If the value passed with the attribute is FALSE, the corresponding flag in Gadget.Flags is cleared. See the *<intuition/intuition.h>* include file or the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

GA_EndGadget, GA_Immediate, GA_RelVerify, GA_FollowMouse, GA_RightBorder, GA_LeftBorder, GA_TopBorder, GA_BottomBorder, GA_ToggleSelect, GA_TabCycle (IS) - These are boolean attributes that correspond to the flags in the object's Gadget.Activation field. If the value passed with the attribute is TRUE, the corresponding flag in Gadget.Activation is set. If the value passed with the attribute is FALSE, the corresponding flag in Gadget.Activation is cleared. See the *<intuition/intuition.h>* include file or the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

GA_Highlight (IS) - This attribute corresponds to the GFLG_GADGHIGHBITS portion of the gadget's Gadget.Flags field. This attribute can be one of four values (from *<intuition/intuition.h>*):

```
GFLG_GADGHCOMP, GFLG_GADGHBOX, GFLG_GADGHIMAGE, GFLG_GADGHNONE
```

See the *<intuition/intuition.h>* include file or the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

GA_SysGType (IS) - This attribute corresponds to the system gadget type portion of the gadget's Gadget.GadgetType fields. This attribute is any *one* of the following flags (from *<intuition/intuition.h>*):

```
GTYP_SIZING, GTYP_WDRAGGING, GTYP_SDRAGGING, GTYP_WUPFRONT, GTYP_SUPFRONT, GTYP_WDOWNBACK,
GTYP_SDOWNBACK, GTYP_CLOSE
```

See the *<intuition/intuition.h>* include file or the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

Class:	propgclass
Superclass:	gadgetclass
Include File:	*<intuition/gadgetclass.h>*

A Boopsi proportional ("prop") gadget. The Boopsi prop gadget is similar to the conventional prop gadget, but extends its function to make it easier to use. The Boopsi prop gadget keeps its current integer value in its PGA_Top attribute.

New Methods:

None.

Changed Methods:

GM_HANDLEINPUT - If the knob position changes sufficiently to change a propgclass object's PGA_Top attribute, the gadget will send an OM_NOTIFY message to itself, which the propgclass dispatcher passes on to the gadgetclass dispatcher for processing (see the rootclass description of OM_NOTIFY and OM_UPDATE for more information).

The OM_NOTIFY message will contain two attribute/value pairs, PGA_Top and GA_ID. While the prop gadget's PGA_Top is in a transitory state (while it is active and the user is moving the prop gadget's knob), the gadget sends interim OM_NOTIFY messages. The interim OM_NOTIFY messages have the OPUF_INTERIM flag of the opUpdate.opu_Flags field set. When the user finishes manipulating the gadget (by letting go of the knob), the gadget sends a final OM_NOTIFY message, which has a cleared OPUF_INTERIM flag.

Attributes:

GA_Image (I) - Propgclass intercepts this gadgetclass attribute before passing it on to gadgetclass. This attribute passes an image for the prop gadget's knob, which gets stored in the propgclass object's Gadget.Image structure. If the propgclass does not get a GA_Image when it creates a prop gadget, the prop gadget's knob defaults to an AUTOKNOB. An AUTOKNOB automatically sizes itself according to how large the range of the gadget is compared to the visible range of the gadget. See the PGA_Visible and PGA_Total attributes for more details.

GA_Border (I) - Propgclass intercepts this gadgetclass attribute to prevent gadgetclass from setting up a border. If an application tries to set this attribute for a propgclass gadget, the prop gadget turns itself into an AUTOKNOB gadget.

GA_Highlight (I) - Propgclass intercepts this gadgetclass attribute before passing it on to gadgetclass. It does this to make sure the highlighting is not set to GADGHBOX. GADGHBOX will be converted to GADGHCOMP. See the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information on the types of gadget highlighting.

Other gadgetclass attributes are passed along to the superclass.

PGA_Freedom (IG) - This attribute tells a propgclass object on which axis the gadget's knob is free to move, the horizontal or the vertical. It is either FREEHORIZ or FREEVERT. The default is FREEVERT.

PGA_NewLook (I) - This is a boolean attribute which corresponds to the PROPNEWLOOK flag PropInfo structure's Flags field (defined in <*intuition/intuition.h*>). If this attribute is TRUE, the new propgclass object will use Release 2 imagery rather than the Release 1.3 imagery.

PGA_Borderless (I) - This is a boolean attribute which corresponds to the BORDERLESS flag of the PropInfo structure's Flags field (defined in <*intuition/intuition.h*>). If this attribute is TRUE, the new propgclass object will not have a border around it. In an AUTOKNOB propgclass gadget, if the PROPNEWLOOK flag is set as well (see the PGA_NewLook attribute), the knob will have a 3D appearance.

PGA_Top (ISGNU)
PGA_Visible, PGA_Total (ISU) - These attributes replace the Pot and Body variables of the Release 1.3 prop gadget. They are based on the use of proportional gadgets to control scrolling text. When scrolling 100 lines of text in a 25 line visible window, you would set PGA_Total to 100, PGA_Visible to 25, and watch PGA_Top run from 0 to 75 (the top line of the last page).

If the user clicks in the prop gadget but not on the knob, the entire knob jumps one "page" (the size of the visible area minus one, PGA_Visible-1). The page jump will leave an overlap of one line, unless the value PGA_Visible is 1, in which case the prop gadget acts as an integer numeric slider, sliding from 0 to PGA_Total - 1.

Note that when PGA_Top changes, the gadget sends itself an OM_NOTIFY message about this attribute change (see the propgclass description of GM_HANDLEINPUT for more information). All three of these attributes have OM_UPDATE access, so they can be controlled from other objects.

Class:	strgclass
Superclass:	gadgetclass
Include File:	<*intuition/gadgetclass.h*>

Intuition compatible string gadgets. The Boopsi string gadget can either be a plain string gadget or an integer string gadget. An integer gadget filters out all characters except those that make up integer values.

New Methods:

None.

Changed Methods:

OM_NEW - This method sets up the string gadget's StringInfo and StringExtend structures. It allocates a buffer if needed and will use shared data buffers for UndoBuffer and WorkBuffer if the MaxChars is less than SG_DEFAULTMAXCHARS (128). Default text pens are: Foreground = 1, Background = 0. See the rootclass description of the OM_NEW method for more details.

Attributes:

STRINGA_LongVal (ISGNU) - This attribute tells strgclass that this gadget is an integer string gadget and the new value of the integer is this attribute's value.

STRINGA_TextVal (ISGNU) - This attribute tells strgclass that this gadget is a plain string gadget. The attribute points to a string which the object copies into the string gadget's current string value buffer.

When a strgclass gadget's internal STRINGA_LongVal or STRINGA_TextVal value changes (usually because the user manipulated the gadget), it sends itself an OM_NOTIFY message. The OM_NOTIFY message will contain two attribute/value pairs, GA_ID and either STRINGA_LongVal or STRINGA_TextVal (depending on what kind of strgclass gadget it is). Strgclass gadgets only send a final OM_NOTIFY message (one's with the OPUF_INTERIM flag of the opUpdate.opu_Flags field cleared).

The remaining strgclass attributes correspond to the flags and fields that the conventional Intuition string gadget uses. See the "STRING GADGET TYPE" section of the "Intuition Gadgets" chapter of the *Amiga ROM Kernel Reference Manual: Libraries* for more information.

STRINGA_MaxChars, STRINGA_Buffer, STRINGA_UndoBuffer, STRINGA_WorkBuffer (I) - Specify various buffers defined for string gadgets and extended string gadgets. If your value of STRINGA_MaxChars is less than SG_DEFAULTMAXCHARS (128 for now), then this class can provide all these buffers for you. Note that UndoBuffer and WorkBuffer can be shared by many separate gadgets, providing they are as large as the largest MaxChars they will encounter.

STRINGA_BufferPos, STRINGA_DispPos (ISU) - The attributes tell the object its cursor and scroll position.

STRINGA_AltKeyMap (IS) - This attribute corresponds to the StringInfo.AltKeyMap field.

STRINGA_Font (IS) - This attributes points to an open TextFont structure that the string gadget uses for rendering its text.

STRINGA_Pens (IS) - Pen numbers, packed as two WORDs into a longword, for rendering gadget text.

STRINGA_ActivePens (IS) - Optional pen numbers, packed as two WORDs into a longword, for rendering gadget text when the gadget is active.

STRINGA_EditHook (I) - Custom string gadget edit hook.

STRINGA_EditModes (IS) - Value taken from flags defined in *<intuition/sghooks.h>* for initial editing modes.

STRINGA_ReplaceMode, STRINGA_FixedFieldMode, STRINGA_NoFilterMode (IS) - These three are independent Boolean equivalents to the individual flags that you can set for STRINGA_EditModes.

STRINGA_Justification (IS) - Takes the values STRINGCENTER, STRINGRIGHT, and STRINGLEFT (which is 0).

STRINGA_ExitHelp (IS) - Set this if you want the gadget to exit when the "Help" key is pressed. Look for a code of 0x5F, the rawkey code for help.

Class: buttongclass
Superclass: gadgetclass
Include File: *<intuition/gadgetclass.h>*

A class of button gadget that continually sends interim OM_UPDATE messages to its target while the user holds down the button. The button sends a final OM_UPDATE message when the user lets go of the button. The imagery for these objects is not built directly into the gadget. Instead, a buttongclass object uses a Boopsi image object, which it gets from its GA_Image attribute.

New Methods:

None.

Changed Methods:

GM_HITTEST - This method gets passed over to the button's image for processing. The button's IM_HITTEST checks for the hit.

GM_HANDLEINPUT - This method continuously issues OM_NOTIFY messages for each IECLASS_TIMER event it gets. The OM_NOTIFY message's OPUF_INTERIM flag (from opUpdate.opu_Flags) is set for all but the final OM_NOTIFY.

The OM_NOTIFY message contains one attribute/value pair, GA_ID. If the pointer is currently over the gadget image, the value of this attribute/value pair is the gadget's actual GA_ID (from the Gadget.GadgetID field). If the pointer isn't over the image, the value is the negative of the gadget's actual GA_ID.

GM_RENDER - All rendering is passed along to the GadgetRender.Image (the GA_Image attribute). This method can tell its image to render in any of four image states: IDS_INACTIVESELECTED, IDS_INACTIVENORMAL, IDS_SELECTED, or IDS_NORMAL.

Attributes:

GA_Image (IS) - This attribute points to the gadget's Boopsi image. Changing this attribute will cause the gadget to refresh its imagery.

Class: frbuttonclass
Superclass: buttongclass
Include File: <*intuition/gadgetclass.h*>

This is a special class of button gadget that puts a Boopsi framing image around some other display element. This display element can be one of three things: plain text from the `GA_Text` attribute, an IntuiText from the `GA_IntuiText` attribute, or an Image from the `GA_LabelImage` attribute.

The user activates the gadget by clicking within the bounds of the gadget's framing image, which it gets from the `GA_Image` attribute. Usually the framing image is an instance of an image class that supports the `IM_FRAMEBOX` method (like frameiclass). If the framing image supports the `IM_FRAMEBOX` method, the frbuttonclass object centers the frame image around the display element. See the imageclass description of `IM_FRAMEBOX` for more information.

New Methods:

None.

Changed Methods:

OM_NEW - When this class creates an object, it sets the object's embedded Gadget.Width and Gadget.Height fields according to the frame image in `GA_Image`. If the `GA_Image` understands the `IM_FRAMEBOX` method, the gadget asks the `GA_Image` what it dimensions would be if it had to surround the display element. If the `GA_Image` does not support `IM_FRAMEBOX`, it just copies the `GA_Image` image's width and height into the Gadget structure.

GM_HITTEST - The gadget delegates this method to the framing image's `IM_HITFRAME` method.

GM_RENDER - For this method, the frbuttonclass object first draws the framing image by sending the image an `IM_DRAWFRAME` message. The object then draws its display element.

Attributes:

GA_Width, GA_Height (S) - These attribute correspond to the gadget's Width and Height fields. If the framing image supports `IM_FRAMEBOX`, changing these resizes the framing image. The framing image re-centers itself around the display element as best it can, and the frbuttonclass gadget re-renders the whole itself.

GA_DrawInfo (I) - This attribute passes a pointer to a valid DrawInfo structure. If the frbuttonclass gadget is going to frame plain text (passed to it in the `GA_Text` attribute), the frbuttonclass gadget requires a DrawInfo structure to properly calculate the dimensions of the text.

GA_Text, GA_IntuiText, GA_LabelImage (IS) - These attributes tell the frbuttonclass object what kind of imagery to use as its display element. See their description in the gadgetclass entry for more information.

Class: groupgclass
Superclass: gadgetclass
Include File: <*intuition/gadgetclass.h*>

This is a class of objects that maintains an internal list of gadgets. Its purpose is to make it easier to layout a group of gadgets. Any gadgets that are a member of a groupgclass object are rendered relative to the groupgclass object's `GA_Left` and `GA_Top` attributes. As new gadgets are added to the groupgclass object, the groupgclass object's dimensions grow to enclose the new gadgets. When the groupgclass object receives an `OM_DISPOSE` message, it not only disposes of itself, it also disposes of all the gadgets in its list. Groupgclass does not support the gadget relative flags (`GA_RelWidth`, `GA_RelHeight`, `GA_RelBottom`, and `GA_RelRight`).

New Methods:

None.

Changed Methods:

OM_SET - This method passes most attributes to the superclass, but propagates changes in position to its members appropriately. Also, `GA_Width` and `GA_Height` are calculated from the position and dimension of the membership.

OM_ADDMEMBER - This method adds a gadget to the group object's list. The group object will increase the size of its select box to include the new gadget's select box. The group object moves the new member to an absolute location (by changing the new member's `GA_Left` and `GA_Top` attributes) relative to the group object's upper-left corner. Note that all members of the groupgclass object will be deleted by `OM_DISPOSE`.

OM_REMMEMBER - This method removes a gadget added to the group object's list with **OM_ADDMEMBER**. Note that all members of the groupgclass object will be deleted by `OM_DISPOSE`.

OM_DISPOSE - This method disposes of the groupgclass object and all its member gadgets.

GM_HITTEST - This method works its way through the list of group members, sending each a **GM_HITTEST** message, looking for the first member in the list that says it has been hit. This member gadget becomes the active member.

GM_RENDER - This method sends a **GM_HITTEST** message to each of its members.

GM_GOACTIVE/GM_GOINACTIVE/GM_HANDLEINPUT - This method passes the message to the active member's dispatcher for processing. For **GM_GOINACTIVE** and **GM_HANDLEINPUT**, the coordinates passed to the member's dispatcher in the message's gpi_Mouse.X and gpi_Mouse.Y fields are translated so that they are relative to the gadget's upper-left corner.

Attributes:

GA_Left, GA_Top (IS) - These attributes correspond to the embedded Gadget.LeftEdge and Gadget.TopEdge fields. Setting these attributes in groupgclass object causes it to change its position as well as the position of each of the gadgets that have been added to the group gadget.

Appendix C
SAMPLE LIBRARY SOURCE CODE

This appendix contains the source code to an Exec library called *sample.library*. It also includes some example code that uses the library. The files include:

makefile	A make file to build the library and examples.
asmsupp.i	Assorted low level assembly support routines used by the example library.
sample_rev.i	The version file generated by *bumprev*.
sample_protos.h	Prototypes for sample.library functions.
sample_pragmas.h	Pragmas for sample.library functions.
samplebase.i	Definition of sample.library base.
samplebase.h	C include file defining sample.library base.
sample.library.asm	Example run-time library source code.
sample_lvos.asm	The .fd file for the sample library.
sample_stubs.asm	Assembler stubs for C interface to library functions.
clibtest.c	C example that calls the sample.library functions.
alibtest.asm	Assembly code example that calls the sample.library functions.

Makefile

```
# MAKEFILE for sample.library, sample.lib, and asm library test program
# define linker command
LINKER = Blink

all: sample.library sample.lib alibtest

sample.library: sample.library.o
      $(LINKER) from sample.library.o to sample.library LIB lib:amiga.lib

sample.lib: sample_stubs.o sample_lvos.o
      JOIN sample_stubs.o sample_lvos.o AS sample.lib

alibtest: alibtest.o
  $(LINKER) from astartup.obj,alibtest.o to alibtest LIB lib:amiga.lib,sample.lib

# assembler command line for Adapt assembler
      HX68 -A $*.asm -iINCLUDE: -o$*.o
```

asmsupp.i

```
**************************************************************************************
*assorted low level assembly support routines used by the Commodore sample Library & Device
**************************************************************************************
CLEAR   MACRO           ;quick way to clear a D register on 68000
        MOVEQ   #0,\1
        ENDM

LINKSYS MACRO           ; link to a library without having to see a _LVO
        MOVE.L  A6,-(SP)
        MOVE.L  \2,A6
        JSR     _LVO\1(A6)
        MOVE.L  (SP)+,A6
        ENDM

CALLSYS MACRO           ; call a library via A6 without having to see _LVO
        JSR     _LVO\1(A6)
        ENDM

XLIB    MACRO           ; define a library reference without the _LVO
        XREF    _LVO\1
        ENDM
;
; Put a message to the serial port at 9600 baud.  Used as so:
;
;     PUTMSG   30,<'%s/Init: called'>
;
; Parameters can be printed out by pushing them on the stack and
; adding the appropriate C printf-style % formatting commands.
;
                XREF    KPutFmt
PUTMSG:         MACRO   * level,msg

                IFGE    INFO_LEVEL-\1

                PEA     subSysName(PC)
                MOVEM.L A0/A1/D0/D1,-(SP)
                LEA     msg\@(pc),A0     ;Point to static format string
                LEA     4*4(SP),A1       ;Point to args
                JSR     KPutFmt
                MOVEM.L (SP)+,D0/D1/A0/A1
                ADDQ.L  #4,SP
                BRA.S   end\@

msg\@           DC.B    \2
                DC.B    10
                DC.B    0
                DS.W    0
end\@
                ENDC
                ENDM
```

sample_rev.i

```
* sample_rev.i version file (generated with the ''bumprev'' command)

VERSION         EQU     37
REVISION        EQU     1
DATE    MACRO
                dc.b    '2.3.92'
        ENDM
VERS    MACRO
                dc.b    'sample 37.1'
        ENDM
VSTRING MACRO
                dc.b    'sample 37.1 (2.3.92)',13,10,0
        ENDM
VERSTAG MACRO
                dc.b    0,'$VER: sample 37.1 (2.3.92)',0
        ENDM
```

sample_protos.h

```
/* sample_protos.h - prototypes for sample.library functions */

#ifndef SAMPLE_PROTOS_H
#define SAMPLE_PROTOS_H

LONG AddThese(LONG,LONG);
LONG Double(LONG);

#endif  /* SAMPLE_PROTOS_H */
```

sample_pragmas.h

```
/* sample_pragmas.h - "sample.library" */
#pragma libcall SampleBase Double 1E 001
#pragma libcall SampleBase AddThese 24 1002
```

samplebase.i

```
* samplebase.i -- definition of sample.library base

   IFND   SAMPLE_BASE_I
SAMPLE_BASE_I SET 1


   IFND  EXEC_TYPES_I
   INCLUDE  "exec/types.i"
   ENDC   ; EXEC_TYPES_I

   IFND  EXEC_LIBRARIES_I
   INCLUDE  "exec/libraries.i"
   ENDC   ; EXEC_LIBRARIES_I

;---------------------------
; library data structures
;---------------------------

;  Note that the library base begins with a library node

   STRUCTURE SampleBase,LIB_SIZE
   UBYTE   sb_Flags
   UBYTE   sb_pad
   ;We are now longword aligned
   ULONG   sb_SysLib
   ULONG   sb_DosLib
   ULONG   sb_SegList
   LABEL   SampleBase_SIZEOF


SAMPLENAME   MACRO
      DC.B   'sample.library',0
      ENDM

   ENDC  ;SAMPLE_BASE_I
```

samplebase.h

```
/* samplebase.h -- C include file defining sample.library base */

#ifndef SAMPLE_BASE_H
#define SAMPLE_BASE_H

#ifndef  EXEC_TYPES_H
#include  <exec/types.h>
#endif   EXEC_TYPES_H

#ifndef  EXEC_LIBRARIES_H
#include <exec/libraries.h>
```

```
#endif   EXEC_LIBRARIES_H

/* library data structures-- Note that the library base begins with a library node */

struct SampleBase {
   struct Library LibNode;
   UBYTE  Flags;
   UBYTE  pad;
   /* We are now longword aligned */
   ULONG  SysLib;
   ULONG  DosLib;
   ULONG  SegList;
   };

#define SAMPLENAME   "sample.library"

#endif  /* SAMPLE_BASE_H */
```

sample.library.asm

```
***************************************************************************************
*   sample.library.asm -- Example run-time library source code
*
*   Assemble and link, without startup code, to create Sample.library,
*     a LIBS: drawer run-time shared library
*
*  Linkage Info:
*  FROM     sample.library.o
*  LIBRARY  LIB:Amiga.lib
*  TO       sample.library
***************************************************************************************

   SECTION   code

   NOLIST
   INCLUDE "exec/types.i"
   INCLUDE "exec/initializers.i"
   INCLUDE "exec/libraries.i"
   INCLUDE "exec/lists.i"
   INCLUDE "exec/alerts.i"
   INCLUDE "exec/resident.i"
   INCLUDE "libraries/dos.i"

   INCLUDE "sampleinclude/asmsupp.i"
   INCLUDE "sampleinclude/samplebase.i"
   INCLUDE "sampleinclude/sample_rev.i"

   LIST

   XDEF   InitTable                      ;------ These don't have to be external but it helps
   XDEF   Open                           ;------ some debuggers to have them globally visible
   XDEF   Close
   XDEF   Expunge
   XDEF   Null
   XDEF   LibName
   XDEF   Double
   XDEF   AddThese

   XREF   _AbsExecBase

   XLIB   OpenLibrary
   XLIB   CloseLibrary
   XLIB   Alert
   XLIB   FreeMem
   XLIB   Remove

   ; The first executable location.  This should return an error in case someone tried to
   ; run you as a program (instead of loading you as a library).
Start:
   MOVEQ   #-1,d0
   rts

;-----------------------------------------------------------------------------------
; A romtag structure.  Both "exec" and "ramlib" look for this structure to discover magic
; constants about you (such as where to start running you from...).  The include file
; sample_rev.i (created by hand or preferable with the developer tool ''bumprev''
; resolves the VERSION, REVISION, and VSTRING.
;-----------------------------------------------------------------------------------
   ; Few people will need a priority and should leave it at zero.  The RT_PRI field is used
   ; in configuring the ROMs.  Use "mods" from wack to look at other romtags in the system.
MYPRI   EQU   0

RomTag:
                ;STRUCTURE RT,0
     DC.W    RTC_MATCHWORD      ; UWORD RT_MATCHWORD
     DC.L    RomTag             ; APTR  RT_MATCHTAG
     DC.L    EndCode            ; APTR  RT_ENDSKIP
     DC.B    RTF_AUTOINIT       ; UBYTE RT_FLAGS
     DC.B    VERSION            ; UBYTE RT_VERSION  (defined in sample_rev.i)
     DC.B    NT_LIBRARY         ; UBYTE RT_TYPE
     DC.B    MYPRI              ; BYTE  RT_PRI
     DC.L    LibName            ; APTR  RT_NAME
     DC.L    IDString           ; APTR  RT_IDSTRING
     DC.L    InitTable          ; APTR  RT_INIT  table for InitResident()

   ; this is the name that the library will have
LibName:   SAMPLENAME
   ; standard name/version/date ID string from bumprev-created sample_rev.i
IDString:  VSTRING

dosName:   DOSNAME

   ; force word alignment
   ds.w   0

   ; The romtag specified that we were "RTF_AUTOINIT".  This means that the RT_INIT
   ; structure member points to one of these tables below.  If the AUTOINIT bit was not
   ; set then RT_INIT would point to a routine to run.

InitTable:
   DC.L   SampleBase_SIZEOF ; size of library base data space
   DC.L   funcTable         ; pointer to function initializers
   DC.L   dataTable         ; pointer to data initializers
   DC.L   initRoutine       ; routine to run


funcTable:

   ;------ standard system routines
   dc.l   Open
   dc.l   Close
   dc.l   Expunge
   dc.l   Null

   ;------ my libraries definitions
   dc.l   Double
   dc.l   AddThese

   ;------ function table end marker
   dc.l   -1


   ; The data table initializes static data structures.  The format is specified in
   ; exec/InitStruct routine's manual pages.  The INITBYTE/INITWORD/INITLONG routines are
   ; in the file "exec/initializers.i".  The first argument is the offset from the library
   ; base for this byte/word/long.  The second argument is the value to put in that cell.
   ; The table is null terminated.
   ; NOTE - LN_TYPE below is a correction - old example had LH_TYPE.

dataTable:
        INITBYTE        LN_TYPE,NT_LIBRARY
        INITLONG        LN_NAME,LibName
        INITBYTE        LIB_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
        INITWORD        LIB_VERSION,VERSION
        INITWORD        LIB_REVISION,REVISION
        INITLONG        LIB_IDSTRING,IDString
        DC.L   0

   ; This routine gets called after the library has been allocated.  The library pointer is
   ; in D0.  The segment list is in A0.  If it returns non-zero then the library will be
   ; linked into the library list.
```

```
initRoutine:

   ;------ get the library pointer into a convenient A register
   move.l   a5,-(sp)
   move.l   d0,a5

   ;------ save a pointer to exec
   move.l   a6,sb_SysLib(a5)

   ;------ save a pointer to our loaded code
   move.l   a0,sb_SegList(a5)

   ;------ open the dos library
   lea   dosName(pc),a1
   CLEAR   d0
   CALLSYS   OpenLibrary

   move.l   d0,sb_DosLib(a5)
   bne.s   1$

   ;------ can't open the dos!  what gives
   ALERT   AG_OpenLib!AO_DOSLib

1$:
   ;------ now build the static data that we need
   ;
   ; put your initialization here...
   ;

   move.l   a5,d0
   move.l   (sp)+,a5
   rts

;----------------------------------------------------------------------------------------
; here begins the system interface commands.  When the user calls OpenLibrary/CloseLibrary/
; RemoveLibrary, this eventually gets translated into a call to the following routines
; (Open/Close/Expunge).  Exec has already put our library pointer in A6 for us.  Exec has
; turned off task switching while in these routines (via Forbid/Permit), so we should not
; take too long in them.
;----------------------------------------------------------------------------------------

   ; Open returns the library pointer in d0 if the open was successful.  If the open failed
   ; then null is returned.  It might fail if we allocated memory on each open, or if only
   ; open application could have the library open at a time...

Open:      ; ( libptr:a6, version:d0 )

   ;------ mark us as having another opener
   addq.w   #1,LIB_OPENCNT(a6)

   ;------ prevent delayed expunges
   bclr   #LIBB_DELEXP,sb_Flags(a6)

   move.l   a6,d0
   rts

   ; There are two different things that might be returned from the Close routine.  If the
   ; library is no longer open and there is a delayed expunge then Close should return the
   ; segment list (as given to Init).  Otherwise close should return NULL.

Close:      ; ( libptr:a6 )

   ;------ set the return value
   CLEAR   d0

   ;------ mark us as having one fewer openers
   subq.w   #1,LIB_OPENCNT(a6)

   ;------ see if there is anyone left with us open
   bne.s   1$

   ;------ see if we have a delayed expunge pending
   btst   #LIBB_DELEXP,sb_Flags(a6)
   beq.s   1$

   ;------ do the expunge
   bsr   Expunge
1$:
   rts

   ; There are two different things that might be returned from the Expunge routine.  If
   ; the library is no longer open then Expunge should return the segment list (as given
   ; to Init).  Otherwise Expunge should set the delayed expunge flag and return NULL.
   ;
   ; One other important note: because Expunge is called from the memory allocator, it may
   ; NEVER Wait() or otherwise take long time to complete.

Expunge:   ; ( libptr: a6 )

   movem.l   d2/a5/a6,-(sp)
   move.l   a6,a5
   move.l   sb_SysLib(a5),a6

   ;------ see if anyone has us open
   tst.w   LIB_OPENCNT(a5)
   beq   1$

   ;------ it is still open.  set the delayed expunge flag
   bset   #LIBB_DELEXP,sb_Flags(a5)
   CLEAR   d0
   bra.s   Expunge_End

1$:
   ;------ go ahead and get rid of us.  Store our seglist in d2
   move.l   sb_SegList(a5),d2

   ;------ unlink from library list
   move.l   a5,a1
   CALLSYS   Remove

   ;
   ; device specific closings here...
   ;

   ;------ close the dos library
   move.l   sb_DosLib(a5),a1
   CALLSYS   CloseLibrary

   ;------ free our memory
   CLEAR   d0
   move.l   a5,a1
   move.w   LIB_NEGSIZE(a5),d0

   sub.l   d0,a1
   add.w   LIB_POSSIZE(a5),d0

   CALLSYS   FreeMem

   ;------ set up our return value
   move.l   d2,d0

Expunge_End:
   movem.l   (sp)+,d2/a5/a6
   rts

Null:
   CLEAR   d0
   rts

;----------------------------------------------------------------------------------------
; Here begins the library specific functions.  Both of these simple functions are entirely
; in assembler, but you can write your functions in C if you wish and interface to them
; here.  If, for instance, the bulk of the AddThese function was written in C, you could
; interface to it as follows:
;
;   - write a C function  addTheseC(n1,n2) and compile it
;   - XDEF _addThese C  in this library code
;   - change the AddThese function code below to:
;       move.l d1,-(sp)      ;push rightmost C arg first
;       move.l d0,-(sp)      ;push other C arg(s), right to left
;       jsr    _addTheseC    ;call the C code
;       addq   #8,sp         ;fix stack
;       rts                  ;return with result in d0
;----------------------------------------------------------------------------------------
```

```
*----- Double(d0)
Double:
   lsl     #1,d0
   rts

*----- AddThese(d0,d1)
AddThese:
   add.l   d1,d0
   rts

   ; EndCode is a marker that show the end of your code.  Make sure it does not span
   ; sections nor is before the rom tag in memory!  It is ok to put it right after the ROM
   ; tag--that way you are always safe.  I put it here because it happens to be the "right"
   ; thing to do, and I know that it is safe in this case.
EndCode:

   END
```

sample_lvos.asm

```
*************************************************************************
* sample_lvos.asm   _LVO definitions
*
*  This is the .fd file for our sample library:
*
* Note - the slash in (D0/D1) means that a stub-maker can use MOVEM.L to
* load these registers from the stack, rather than using a separate MOVE
* instruction for each register. Alternately, something like  (A0,D2) would
* show that a separate MOVE instruction is needed for each load.
*
*       ##base _SampleBase
*       ##bias 30
*       ##public
*       Double(n1)(D0)
*       AddThese(n1,n2)(D0/D1)
*       ##end
*
* After assembling,
*   JOIN sample_stubs.o sample_lvos.o AS sample.lib
*
* Apps LINK with LIBRARY sample.lib when calling sample.library functions
*************************************************************************

   INCLUDE  "exec/types.i"
   INCLUDE  "exec/libraries.i"

          SECTION data

*----- LIBINIT initializes an LVO value to -30 to skip the first four
*----- 6-byte required library vectors (Open, Expunge, etc)

             LIBINIT

*----- LIBDEF assigns the current LVO value to a label, and then
*----- bumps the LVO value by -6 in preparation for next LVO label

*----- This assigns the value -30 to our first _LVO label

             LIBDEF      _LVODouble     ;-30
             XDEF        _LVODouble

*----- The value -30-6 is asigned to our second _LVO label

             LIBDEF      _LVOAddThese   ;-36
             XDEF        _LVOAddThese

             END
```

sample_stubs.asm

```
**********************************************************************************************
* sample_stubs.asm
*
* Stubs match this .fd file:
*
*       ##base _SampleBase
*       ##bias 30
*       ##public
*       Double(n1)(D0)
*       AddThese(n1,n2)(D0/D1)
*       ##end
*
* After assembling,
*   JOIN sample_stubs.o sample_lvos.o AS sample.lib
*
* Apps LINK with LIBRARY sample.lib when calling sample.library functions
*
* If you put all of your stubs in one file, as shown here, then ALL of the stubs will be
* linked into an application that references one stub.  For larger libraries, you should
* place each stub in a separate assembler file, assemble them each separately, then join
* all of the .o's together. That will allow each stub to be independently pulled into the
* application that links with the .lib.
**********************************************************************************************
   INCLUDE  "exec/types.i"
   INCLUDE  "exec/libraries.i"

          section code

*------ Caller declares and initializes SampleBase in their C code

            XREF        _SampleBase

*------ Must externally reference the _LVO labels defined in samplelib_lvos

            XREF        _LVODouble
            XREF        _LVOAddThese

*------ Make C function stubs available to caller

            XDEF        _Double
            XDEF        _AddThese

*------- These stubs move C args from stack to appropriate registers,
*------- call the library function, and return result in d0

_Double:
            MOVE.L      A6,-(SP)            ;Save register(s)
            MOVE.L      8(SP),D0            ;Copy param to register
            MOVE.L      _SampleBase,A6      ;Library base to A6
            JSR         _LVODouble(A6)      ;Go to real routine
            MOVE.L      (SP)+,A6            ;Restore register(s)
            RTS

_AddThese:
            MOVE.L      A6,-(SP)            ;Save register(s)
            MOVEM.L     8(SP),D0/D1         ;Copy params to registers
                                            ;8(SP)  goes into D0
                                            ;12(SP) goes into D1
            MOVE.L      _SampleBase,A6      ;Library base to A6
            JSR         _LVOAddThese(A6)    ;Go to real routine
            MOVE.L      (SP)+,A6            ;Restore register(s)
            RTS

          END
```

clibtest.c

```
;/* clibtest.c--Calls the Sample.library functions (execute to compile with Lattice 5.10a)
LC -b1 -cfistq -v -y -j73 clibtest.c
Blink FROM LIB:c.o,clibtest.o TO clibtest LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
; note - you must also link with sample.lib if not using sample_pragmas.h
*/

#include <exec/types.h>
#include <exec/libraries.h>
#include <libraries/dos.h>

#include <clib/exec_protos.h>

#include <stdlib.h>
#include <stdio.h>

#include "sampleinclude/samplebase.h"
#include "sampleinclude/sample_protos.h"
#include "sampleinclude/sample_pragmas.h"

#ifdef LATTICE
int CXBRK(void)     { return(0); }      /* Disable Lattice CTRL/C handling */
void chkabort(void) { return; }         /* really */
#endif

struct Library *SampleBase = NULL;

void main(int argc, char **argv)
   {
   LONG n;

   /* Open sample.library */
   if(!(SampleBase=OpenLibrary("sample.library",0)))
      {
      printf("Can't open sample.library\n");
      exit(RETURN_FAIL);
      }
   /* Print library name, ID string, version, revision */
   printf("%s  Version=%ld   Revision=%ld  IdString: %s\n",
             SampleBase->lib_Node.ln_Name, SampleBase->lib_Version,
             SampleBase->lib_Revision, SampleBase->lib_IdString);

   /* Call the two functions */
   n = Double(-7);
   printf("Function Double(-7) returned %ld\n", n);
   n = AddThese(21,4);
   printf("Function AddThese(21,4) returned %ld\n", n);

   CloseLibrary(SampleBase);
   exit(RETURN_OK);
   }
```

alibtest.asm

```
***********************************************************************
*   alibtest.asm -- Asm example that calls the Sample.library functions
*
*   Linkage Info:
*   FROM     Astartup.obj, alibtest.o
*   LIBRARY  LIB:amiga.lib, LIB:sample.lib
*   TO       ALibTest
***********************************************************************

   INCLUDE   "exec/types.i"
   INCLUDE   "exec/libraries.i"

   INCLUDE   "sampleinclude/asmsupp.i"
   INCLUDE   "sampleinclude/samplebase.i"

ABSEXECBASE  EQU  4

   XDEF   _main

   XREF   _printf
   XREF   _LVODouble
   XREF   _LVOAddThese

   XLIB   OpenLibrary
   XLIB   CloseLibrary

        section code
_main:
   ;------ open the test library: this will bring it in from disk
   move.l   ABSEXECBASE,a6
   lea      sampleName(pc),a1
   moveq    #0,d0
   jsr      _LVOOpenLibrary(a6)

   tst.l    d0
   bne.s    1$

   ;------ couldn't find the library
   pea      sampleName(pc)
   pea      nolibmsg(pc)
   jsr      _printf
   addq.l   #8,sp

   bra      main_end

1$:
   move.l   d0,a6    ;sample.library base to a6

   ;------ print the library name, version, and revision
   clr.l    d0
   move.w   LIB_REVISION(a6),d0
   move.l   d0,-(sp)
   move.w   LIB_VERSION(a6),d0
   move.l   d0,-(sp)
   move.l   LN_NAME(a6),-(sp)
   pea      verRevMsg(pc)
   jsr      _printf            ;call Amiga.lib printf
   adda.l   #16,sp             ;fix 4 long stack pushes

   ;------ call the first test function
   moveq    #-7,d0
   jsr      _LVODouble(a6)
   move.l   d0,-(sp)
   pea      doubleMsg(pc)
   jsr      _printf
   lea      8(sp),sp           ;fix 2 long stack pushes

   ;------ call the second test function
   moveq    #21,d0
   moveq    #4,d1
   jsr      _LVOAddThese(a6)
   move.l   d0,-(sp)
   pea      addTheseMsg(pc)
   jsr      _printf
   lea      8(sp),sp

   ;------ close the library
   move.l   a6,a1
   move.l   ABSEXECBASE,a6
   jsr      _LVOCloseLibrary(a6)

main_end:
            rts

sampleName:  SAMPLENAME
nolibmsg:    dc.b   'can not open library "%s"',10,0
doubleMsg:   dc.b   'Function Double(-7) returned %ld',10,0
addTheseMsg: dc.b   'Function AddThese(21,4) returned %ld',10,0
verRevMsg:   dc.b   '%s   Version %ld   Revision %ld',10,0
   END
```

Appendix D
Troubleshooting Your Software

Many Amiga programming errors have classic symptoms. This guide will help you to eliminate or avoid these problems in your software.

Audio—Corrupted Samples

The bit data for audio samples *must* be in Chip RAM. Check your compiler manual for directives or flags which will place your audio sample data in Chip RAM. Or dynamically allocate Chip RAM and copy or load the audio sample there.

Character Input/Output Problems

RAWKEY users must be aware that RAWKEY codes can be different letters or symbols on national keyboards. If you need to use RAWKEY, run the codes through **RawKeyConvert()** (see the ''Intuition Mouse and Keyboard'' chapter) to get proper translation to correct ASCII codes. Improper display or processing of high-ASCII international characters can be caused by incorrect **tolower()/toupper()**, or by sign extension of character values when switched on or assigned into larger size variables. Use unsigned variables such as UBYTE (not char) for strings and characters whenever possible. Internationally correct string functions are provided in the 2.0 utility.library.

CLI Error Message Problems

Improper error messages are caused by calling **exit(n)** with an invalid or missing return value *n*. Assembler programmers using startup code should jump to the startup code's *_exit* with a valid return value on the stack. Programs without startup code should return with a valid value in D0. Valid return values such as RETURN_OK, RETURN_WARN, RETURN_FAIL are defined in *<dos/dos.h>* and *<dos/dos.i>*. Values outside of these ranges (-1 for instance) can cause invalid CLI error messages such as "not an object module". Useful hint—if your program is called from a script, your valid return value can be conditionally branched on in the script (i.e., call program, then perform actions based on IF WARN or IF NOT WARN). RETURN_FAIL will cause the script to stop if a normal FAILAT value is being used in script.

CLI Won't Close on RUN

A CLI can't close if a program has a **Lock()** on the CLI input or output stream ("*"). If your program is RUN >NIL: from a CLI, that CLI should be able to close unless your code or your compiler's startup code explicitly opens "*".

Crashes and Memory Corruption

Memory corruption, address errors, and illegal instruction errors are generally caused by use of an uninitialized, incorrectly initialized, or already freed/closed pointer or memory. You may be using the pointer directly, or it may be one that you placed (or forgot to place) in a structure passed to system calls. Or you may be overwriting one of your arrays, or accidentally modifying or incrementing a pointer later used in a free/close. Be sure to test the return of all open/allocation type functions before using the result, and only close/free things that you successfully opened/allocated. Use watchdog/torture utilities such as *Enforcer* and *MungWall* in combination to catch use of uninitialized pointers or freed memory, and other memory misuse problems. Use the debugging tool *TNT* to get additional debugging information instead of a Software Error requester. You may also be overflowing your stack—your compiler's stack checking option may be able to catch this. Cut stack usage by dynamically allocating large structures, buffers, and arrays which are currently defined inside your functions.

Corruption or crashes can also be caused by passing wrong or missing arguments to a system call (for example **SetAPen(3)** or **SetAPen(win,3)**, instead of **SetAPen(rp,3)**). C programmers should use function prototypes to catch such errors. If using short integers be sure to explicitly type long constants as long (e.g., 42L). (For example, with short ints, 1 << 17 may become zero). If corruption is occurring during exit, use **printf**() (or **kprintf**(), etc.) with **Delay**(n) to slow down your cleanup and broadcast each step. A bad pointer that causes a system crash will often be reported as an standard 680x0 processor exception $00000003 or 4, or less often a number in the range of $00000006-B. Or an Amiga-specific alert number may result. See *<exec/alerts.h>* for Amiga-specific alert numbers. Also see "Crashes—After Exit" below.

Crashes—After Exit

If this only happens when you start your program from Workbench, then you are probably **UnLocking**() one of the **WBStartup** message **wa_Locks**, or **UnLocking()** the **Lock()** returned from an initial **CurrentDir**() call. If you **CurrentDir**(), save the lock returned initially, and **CurrentDir()** back to it before you exit. Only **UnLock**() locks that *you* created.

If you are crashing from both Workbench and CLI, and you are only crashing *after* exit, then you are probably either freeing/closing something twice, or freeing/closing something your did not actually allocate/open, or you may be leaving an outstanding device I/O request or other wakeup request. You must abort and **WaitIO**() any outstanding I/O requests before you free things and exit (see the Autodocs for your device, and for Exec **AbortIO**() and **WaitIO**()). Similar problems can be caused by deleting a subtask that might be in a **WaitTOF**(). Only delete subtasks when you are sure they are in a safe state such as **Wait(0L)**.

Crashes—Only on 68000 and 68010

This can be caused by illegal instructions (80000000.00000004) such as new 68020/30/40 instructions or inline 68881/882 code. But this is usually caused by a word or longword access at an odd address. This is legal on the 68020 and above, but will generate an Address Error (80000000.00000003) on a 68000 or 68010. This can be caused by using uninitialized pointers, using freed memory, or using system structures improperly (for example, referencing into **IntuiMessage->IAddress** as a struct **Gadget *** on a non-Gadget message).

Crashes—Only on 68040

Because of the instruction pipelining of the 68040, it is very difficult to recover from a bus error. If your program has an "*Enforcer* hit" (i.e., an illegal reference to memory), the resulting 68040 processor bus error will probably crash the machine. Use *Enforcer* (on an '030) to track down your problems, then correct them.

Crashes—Subtasks, Interrupts

If part of your code runs on a different stack or the system stack, you must turn off compiler stack-checking options. If part of your code is called directly by the system or by other tasks, you must use long code/long data or use special compiler flags or options to assure that the correct base registers are set up for your subtask or interrupt code.

Crashes—Window Related

Be careful not to **CloseWindow()** a window during a *while(msg=GetMsg(...))* loop on that window's port (next **GetMsg()** would be on freed pointer). Also, use **ModifyIDCMP(NULL)** with care, especially if using one port with multiple windows. Be sure to **ClearMenuStrip()** any menus before closing a window, and do not free items such as dynamically allocated gadgets and menus while they are attached to a window. Do not reference an **IntuiMessage**'s **IAddress** field as a structure pointer of any kind before determining it *is* a structure pointer (this depends on the **Class** of the **IntuiMessage**). If a crash or problem only occurs when opening a window after extended use of your program, check to make sure that your program is properly freeing up signals allocated indirectly by **CreatePort()**, **OpenWindow()** or **ModifyIDCMP()**.

Crashes—Workbench Only

If you are crashing near the first DOS call, either your stack is too small or your startup code did not **GetMsg()** the **WBStartup** message from the process message port. If your program crashes during execution or during your exit procedure only when started from Workbench, and your startup opens no **stdio** window or NIL: file handles for WB programs, then make sure you are not writing anything to **stdout** (**printf**(), etc.) when started from WB (*argc==0*). See also "Crashes—After Exit".

Device-related Problems

Device-related problems may caused by: improperly initialized port or I/O request structures (use **CreatePort()** and **CreateExtIO()**); use of a too-small I/O request (see the device's <.h> files and Autodocs for information on the required type of I/O request); re-use of an I/O request before it has returned from the device (use the debugging tool *IO_Torture* to catch this); failure to abort and wait for an outstanding device request before exiting; waiting on a signal/port/message allocated by a different task.

Disk Icon Won't Go Away

This occurs when a program leaves a **Lock()** on one or more of a disk's files or directories. A memory loss of exactly 24 bytes is usually **Lock()** which has not been **UnLocked()**.

DOS-related Problems

In general, any dos.library function which fills in a structure for you (for example, **Examine()**), requires that the structure be longword aligned. In most cases, the only way to insure longword alignment in C is to dynamically allocate the structure. Unless documented otherwise, dos.library functions may only be called from a process, not from a task. Also note that a process's **pr_MsgPort** is intended for the exclusive use of dos.library. (The port may be used to receive a **WbStartup** message as long as the message is **GetMsg()**'d from the port before DOS is used.

Fails only on 68020/30

The following programming practices can cause this problem: using the upper bytes of addresses as flags; doing signed math on addresses; self-modifying code; using the MOVE SR assembler instruction (use Exec **GetCC()** instead); software delay loops; assumptions about the order in which asynchronous tasks will finish. The following differences in 68020/30 can cause problems: data and/or instruction caches must be flushed if data or code is changed by DMA or other non-processor modification; different exception stack frame; interrupt autovectors may be moved by VBR; 68020/30 CLR instruction does a single write access unlike the 68000 CLR instruction which does a separate read and write access (this might affect a read-triggered register in I/O space—use MOVE instead).

Fails only on 68000

The following programming practices can be the cause of this problem: software delay loops; word or longword access of an odd address (illegal on the 68000). Note that this can occur under 2.0 if you reference **IntuiMessage->IAddress** as a structure pointer without first determining that the **IntuiMessage**'s **Class** is defined as having a structure pointer in its **IAddress**; use of the assembler CLR instruction on a hardware register which is triggered by any access. The 68000 CLR instruction performs two accesses (read and write) while 68020/30 CLR does a single write access. Use MOVE instead; assumptions about the order in which asynchronous tasks will finish; use of compiler flags which have generated inline 68881/68882 math coprocessor instructions or 68020/30 specific code.

Fails only on Older ROMs or Older WB

This can be caused by asking for a library version higher than you need (Do *not* use the #define LIBRARY_VERSION when compiling!). Can also be caused by calling functions or using structures which do not exist in the older version of the operating system. Ask for the lowest version which provides the functions you need (usually 33), and exit gracefully and informatively if an **OpenLibrary**() fails (returns NULL). Or code conditionally to only use new functions and structures if the available library's **lib->Version** supports them.

Fails only on Newer ROMs or Newer WB

This should not happen with proper programming. Possible causes include: running too close to your stack limits or the memory limits of a base machine (newer versions of the operating system may use slightly more stack in system calls, and usually use more free memory); using system functions improperly; not testing function return values; improper register or condition code handling in assembler code. Remember that result, if any, is returned in D0, and condition codes and D1/A0/A1 are undefined after a system call; using improperly initialized pointers; trashing memory; assuming something (such as a flag) is B if it is not A; failing to initialize formerly reserved structure fields to zero; violating Amiga programming guidelines (for example: depending on or poking private system structures, jumping into ROM, depending on undocumented or unsupported behaviors); failure to read the function Autodocs.

See Appendix E, "Release 2 Compatibility", for more information on 2.0 compatibility problem areas.

Fails only on Chip-RAM-Only Machines

Caused by specifically asking for or requiring MEMF_FAST memory. If you don't need Chip RAM, ask for memory type 0L, or MEMF_CLEAR, or MEMF_PUBLIC|MEMF_CLEAR as applicable. If there is Fast memory available, you will be given Fast memory. If not, you will get Chip RAM. May also be caused by trackdisk-level loading of code or data over important system memory or structures which might reside in low Chip memory on a Chip-RAM-Only machine.

Fails only on machines with Fast RAM

Data and buffers which will be accessed directly by the custom chips *must* be in Chip RAM. This includes bitplanes (use **OpenScreen()** or **AllocRaster()**), audio samples, trackdisk buffers, and the graphic image data for sprites, pointers, bobs, images, gadgets, etc. Use compiler or linker flags to force Chip RAM loading of any initialized data needing to be in Chip RAM, or dynamically allocate Chip RAM and copy any initialization data there.

Fails only with Enhanced Chips

Usually caused by writing or reading addresses past the end of older custom chips, or writing something other than 0 (zero) to bits which are undefined in older chip registers, or failing to mask out undefined bits when interpreting the value read from a chip register. Note that system copper lists are different under 2.0 when ECS chips are present. See "Fails only on Chip-RAM-Only Machines".

Fireworks

A dazzling pyrotechnic video display is caused by trashing or freeing a copper list which is in use, or trashing the pointers to the copper list. If you aren't messing with copper lists, see "Crashes and Memory Corruption".

Graphics—Corrupted Images

The bit data for graphic images such as sprites, pointers, bobs, and gadgets *must* be in Chip RAM. Check your compiler manual for directives or flags which will place your graphic image data in Chip RAM. Or dynamically allocate Chip RAM and copy them there.

Hang—One Program Only

Program hangs are generally caused by **Wait()**ing on the wrong signal bits, on the wrong port, on the wrong message, or on some other event that will never occur. This can occur if the event you are waiting on is not coming, or if one task tries to **Wait()**, **WaitPort()**, or **WaitIO()** on a signal, port, or window that was created by a different task. Both **WaitIO()** and **WaitPort()** can call **Wait()**, and you cannot **Wait()** on another task's signals. Hangs can also be caused by verify deadlocks. Be sure to turn off all Intuition verify messages (such as MENUVERIFY) before calling **AutoRequest()** or doing disk access.

Hang—Whole System

This is generally caused by a **Disable()** without a corresponding **Enable()**. It can also be caused by memory corruption, especially corruption of low memory. See "Crashes and Memory Corruption".

Memory Loss

First determine that your program is actually causing a memory loss. It is important to boot with a standard Workbench because a number of third party items such as some background utilities, shells, and network handlers dynamically allocate and free pieces of memory. Open a Shell for memory checking, and a Shell or Workbench drawer for starting your program. Arrange windows so that all are accessible, and so that no window rearrangement will be needed to run your program.

In the Shell, type *Avail FLUSH<RET>* several times (2.0 option). This will flush all non-open disk-loaded fonts, devices, etc., from memory. Note the amount of free memory. Now without rearranging any windows, start your program and use all of your program features. Exit your program, wait a few seconds, then type *Avail FLUSH<RET>* several times. Note the amount of free memory. If this matches the first value you noted, your program is fine, and is not causing a memory loss.

If memory was actually lost, and your program can be run from CLI or Workbench, then try the above procedure with both methods of starting your program. Note that under 2.0, there will be a slight permanent (until reboot) memory usage of about 672 bytes when the audio.device or narrator.device is first opened. See "Memory Loss—CLI Only" and "Memory Loss—WB Only" if appropriate. If you lose memory from both WB and CLI, then check all of the open/alloc/get/create/lock type calls in your code, and make sure that there is a matching close/free/delete/unlock type call for each of them (note—there are a few system calls that have or require no corresponding free—check the Autodocs). Generally, the close/free/delete/unlock calls should be in opposite order of the allocations.

If you are losing a fixed small amount of memory, look for a structure of that size in the Structure Offsets listing in the *Amiga ROM Kernel Reference Manual: Includes and Autodocs.* For example, a loss of exactly 24 bytes is probably a **Lock()** which has not been **UnLocked()**. If you are using **ScrollRaster()**, be aware that **ScrollRaster()** left or right in a Superbitmap window with no **TmpRas** will lose memory under 1.3 (workaround—attach a **TmpRas**). If you lose much more memory when started from Workbench, make sure your program is not using **Exit(n)**. This would bypass startup code cleanups and prevent a Workbench-loaded program from being unloaded. Use **exit(n)** instead.

Memory Loss—CLI Only

Make sure you are testing in a standard environment. Some third-party shells dynamically allocate history buffers, or cause other memory fluctuations. Also, if your program executes different code when started from CLI, check that code and its cleanup. And check your *startup.asm* if you wrote your own.

Memory Loss—Ctrl-C Exit Only

You have Amiga-specific resources opened or allocated and you have not disabled your compiler's automatic Ctrl-C handling (causing all of *your* program cleanups to be skipped). Disable the compiler's Ctrl-C handling and handle Ctrl-C (SIGBREAKF_CTRL_C) yourself.

Memory Loss—During Execution

A continuing memory loss during execution can be caused by failure to keep up with voluminous **IDCMP** messages such as **MOUSEMOVE** messages. Intuition cannot re-use **IDCMP** message blocks until you **ReplyMsg()** them. If your window's allotted message blocks are all in use, new sets will be allocated and not freed till the window is closed. Continuing memory losses can also be caused by a program loop containing an allocation-type call without a corresponding free.

Memory Loss—Workbench Only

Commonly, this is caused by a failure of your code to unload after you exit. Make sure that your code is being linked with a standard correct startup module, and do *not* use the **Exit(n)** function to exit your program. This function will bypass your startup code's cleanup, including its **ReplyMsg()** of the **WBStartup** message (which would signal Workbench to unload your program from memory). You should exit via either **exit(n)** where *n* is a valid DOS error code such as RETURN_OK (<*dos/libraries.h*>), or via final ''}'' or return. Assembler programmers using startup code can JMP to **_exit** with a long return value on stack, or use the RTS instruction.

Menu Problems

A flickering menu is caused by leaving a pixel or more space between menu subitems when designing your menu. Crashing after browsing a menu (looking at menu without selecting any items) is caused by not properly handling MENUNULL select messages. Multiple selection not working is caused by not handling NextSelect properly. See the ''Intuition Menus'' chapter.

Out-of-Sync Response to Input

Caused by failing to handle all received signals or all possible messages after a **Wait()** or**WaitPort()** call. More than one event or message may have caused your program to awakened. Check the signals returned by **Wait()** and act on every one that is set. At ports which may have more than one message (for instance, a window's IDCMP port), you must handle the messages in a *while(msg=GetMsg(...))* loop.

Performance Loss in Other Processes

This is often caused by a one program doing one or more of the following: busy waiting or polling; running at a higher priority; doing lengthy **Forbids()**, **Disables()**, or interrupts.

Performance Loss—On A3000

If your program has ''*Enforcer* hits'' (i.e., illegal references to memory caused by improperly initialized pointers), this will cause Bus Errors. The A3000 bus error handler contains a built-in delay to let the bus settle. If you have many enforcer hits, this could slow your program down substantially.

Trackdisk Data not Transferred

Make sure your trackdisk buffers are in Chip RAM under 1.3 and lower versions of the operating system.

Windows—Borders Flicker after Resize

Set the NOCAREREFESH flag. Even SMART_REFRESH windows may generate refresh events if there is a sizing gadget. If you don't have specific code to handle this, you must set the NOCAREREFRESH flag. If you do have refresh code, be sure to use the **Begin()/EndRefresh()** calls. Failure to do one or the other will leave Intuition in an intermediate state, and slow down operation for all windows on the screen.

Windows—Visual Problems

Many visual problems in windows can be caused by improper font specification or improper setting of gadget flags. See the Appendix E on "Release 2 Compatibility" for detailed information on common problems.

GENERAL DEBUGGING TECHNIQUES

Narrow the search

Use methodical testing procedures, and debugging messages if necessary, to locate the problem area. Low level code can be debugged using **kprintf()** serial (or **dprintf()** parallel) messages. Check the initial values, allocation, use, and freeing of all pointers and structures used in the problem area. Check that all of your system and internal function calls pass correct initialized arguments, and that all possible error returns are checked for and handled.

Isolate the problem

If errors cannot be found, simplify your code to the smallest possible example that still functions. Often you will find that this smallest example will not have the problem. If so, add back the other features of your code until the problem reappears, then debug that section.

Use debugging tools

A variety of debugging tools are available to help locate faulty code. Some of these are source level and other debuggers, crash interceptors, vital watchdog and memory invalidation tools like *Enforcer* and *MungWall*.

A FINAL WORD ABOUT TESTING

Test your program with memory watchdog and invalidation tools on a wide variety of systems and configurations. Programs with coding errors may appear to work properly on one or more configurations, but may fail or cause fatal problems on another. Make sure that your code is tested on both a 68000 and a 68020/30, on machines with and without Fast RAM, and on machines with and without enhanced chips. Test all of your program functions on every machine.

Test all error and abort code. A program with missing error checks or unsafe cleanup might work fine when all of the items it opens or allocates are available, but may fail fatally when an error or problem is encountered. Try your code with missing files, filenames with spaces, incorrect filenames, cancelled requesters, Ctrl-C, missing libraries or devices, low memory, missing hardware, etc.

Test all of your text input functions with high-ASCII characters (such as the character produced by pressing Alt-F then "A"). Note that RAWKEY codes can be different keyboard characters on national keyboards (higher levels of keyboard input are automatically translated to the proper characters). If your program will be distributed internationally, support and take advantage of the additional screen lines available on a PAL system. Enhanced Agnus chip machines may be switched to be PAL or NTSC via motherboard jumper J102 in A2000s and jumper J200 in A3000s. Note that a base PAL machine will have less memory free due to the larger display size.

Write good code. Test it. Then make it great.

Appendix E
RELEASE 2 COMPATIBILITY

If you are developing new software or updating older software, you need to avoid compatibility traps. This comprehensive list of Release 2 compatibility problem areas can help you avoid and diagnose compatibility problems In addition, you should refer to the ''Amiga Programming Guidelines'' listed in Chapter 1.

General Compatibility Problem Areas

The following improper Amiga programming practices are likely to fail on new ROMs or hardware.

- Requiring all free RAM.
- Overwriting memory allocations. With 32-bit addresses, a 1-byte overwrite of a string array can wipe out the high byte of a pointer or stack return address. This bug could go unnoticed on a 24-bit address machine (e.g., A500) but crash the system or cause other problems on an A3000.
- Improper flags or garbage in system structures. A bit that means nothing under one OS may drastically change the behavior of a function in a newer version of the OS. Clear structures before using, and use correct flags.
- Misuse of function return values. Use function prototypes and read the Autodocs for the functions you are using. Some system functions return just success or failure, or nothing at all (void). In such cases, the value which the function happens to return must not be used except as it is documented.
- Depending on unsupported side effects or undocumented behavior. Be sure to read the Autodocs, include file comments and other documentation.
- Assuming current choices, configurations or initial values. If the current possibilities are A, B, or C, do not assume C if it isn't A or B. Check specifically for the choices currently implemented, and provide default behavior for unexpected values.

Amiga debugging tools such as *Enforcer*, *Mungwall* and *Scratch* can find many program bugs that may affect compatibility. A program that is *Enforcer/Mungwall/Scratch* clean stands a much better chance of working well under current and future versions of the OS.

Release 2 Changes That Can Affect Compatibility

There are several areas where Relase 2 OS changes and enhancements can cause compatibility problems for some software.

EXEC

- Do not jump to location $FC0002—the start of the ROM under 1.3—as part of performing a system RESET. The 2.04 Kickstart ROM has a temporary compatibility hack called "Kickety-Split" which is a redirecting jump at $FC0002. This hack does not appear on the A3000 ROM and due to space considerations will not appear on future machines.

- Everything has moved.

- The Supervisor stack is not in the same place as it was under 1.3. This has caused problems for some games that completely take over the Amiga. If your program goes into Supervisor mode, you must either respect allocated memory or provide your own Supervisor stack when taking over the machine.

- **ExecBase** is moved to expansion memory if possible. Before, **ExecBase** would only end up in one of two fixed locations. Now, ColdCapture may be called after expansion memory has been configured.

- Exception/Interrupt vectors may move. This means the 68010 and above Vector Base Register (VBR) may contain a non-zero value. Poking assumed low memory vector addresses may have no effect. You must read the VBR on 68010 and above to find the base.

- No longer tolerant of wild **Forbid()** counts. Under 1.3, sometimes this bug could go unnoticed. Make sure that all **Forbid()**s are matched with one and only one **Permit()**(and vice versa).

- When an Exec device gets an **IORequest**, it must validate **io_Command**. If the **io_Command** is 0 or out of range, the device must return IOERR_NOCMD and take no other action. The filesystem now sends new commands and expects older devices to properly ignore them.

- A fix to task-switching in Release 2 allows a busy task to properly regain the processor after an interrupt until either its quantum (4 vertical blanks) is up or a higher priority task preempts it. This can dramatically change the behavior of multitask programs where one task busyloops while another same-priority task **Wait()**s. See "Task Switching" in the "Additional Information" section below.

EXPANSION

- **ExpansionBase** is private - use **FindConfigDev()**.

- Memory from contiguous cards of the same memory type is automatically merged into one memory pool.

STRAP

- Romboot.library is gone.
- Audio.device cannot be **OpenDevice**()ed by a boot block program. See "Audio Device" below.
- Boot from other floppies (+5,-10,-20,-30) is possible.
- Undocumented system stack and register usage at Diag and Boot time have changed.

DOS

- DOS is now written in C and assembler, not BCPL. The BCPL compiler artifact which caused D0 function results to also be in D1 is gone. System patches in Release 2 that return some DOS function results in both D0 and D1 are not guaranteed to remain in the next release. Fix your programs! Use *Scratch* to find these problems in your code.
- DOS now has a real library base with normal LVO vectors.
- Stack usage has all changed (variables, direction).
- New packet and lock types. Make sure you are not passing stack garbage for the second argument to **Lock**().
- Process structure is bigger. "Rolling your own" **Process** structure from a **Task fails**. Use dos.library **System**() or **NewCreateProc**().
- Unless documented otherwise, you must be a process to call DOS functions. DOS function dependence on special process structures can change with OS revisions.

AUDIO DEVICE

- Now not initialized until used. This means low memory open failure is possible. Check your return values from **OpenDevice**(). This also means audio.device cannot be opened during 2.0 Strap unless **InitResident**()ed first. If **OpenDevice**() of audio.device fails during strap, you must **FindResident**()/**InitResident**() audio.device, and then try **OpenDevice**() again. There will be a small memory loss (until reboot) generated by the first opener of audio.device or narrator.device (memory used in building of audio.device's base).

GAMEPORT DEVICE

- Initial state of hardware lines may differ.

SERIAL DEVICE

- Clears **io_Device** on **CloseDevice**() (since 1.3.2)

TIMER DEVICE

- The most common mistake programmers make with timer.device is to send off a particular **timerequest** before the previous use of that **timerequest** has completed. Use *IO_Torture* to catch this.
- IO_QUICK requests may be deferred and be replied as documented.
- VBLANK timer requests, as documented, now wait at least as long as the full number of VBlanks you asked for. Previously, a partial vertical blank could count towards your requested number. The new behavior is more correct and matches the docs, but it can cause VBlank requests to now take up to 1 VBlank longer under 2.0 as compared to 1.3. For example, a 1/10 second request, may take 6-7 Vblanks instead of 5-6 VBlanks, or about 15% longer.

TRACKDISK DEVICE

- Private trackdisk structures have changed. See *trackdisk.doc* for a compatible REMCHANGEINT.
- Buffer is freeable, so low memory open failure is possible.
- Do not disable interrupts (any of them), then expect trackdisk to function while they are disabled.

CIA TIMERS

- System use of CIA timers has changed. Don't assume how they're used.
- Don't depend on initial values of CIA registers.
- Don't mess with **CIABase**. Use cia.resource.
- If your code requires hardware level CIA timers, allocate the timers using cia.resource **AddICRVector()**! This is very important. Operating system usage of the CIA timers has changed. The new 2.0 timer.device (''Jumpy the Magic Timer Device'') will try to jump to different CIAs so programs that properly allocate timers will have a better chance of getting what they want. If possible, be flexible and design your code to work with whatever timer you can successfully allocate.
- OS usage of INT6 is increasing. Do not totally take over INT6, and do not terminate the server chain if an interrupt is not for you.

OTHER HARDWARE ISSUES

- Battery-backed clock is different on A3000. Use battclock.resource to access the real-time clock if battclock.resource can be opened.
- A 68030 hardware characteristic causes longword-aligned longword writes to allocate a valid entry in the data cache, even if the hardware area shouldn't be cached. This can cause problems for I/O registers and shared memory devices. To solve this: 1) don't do that 2) flush the cache or 3) use Enforcer Quiet. See the Motorola 68030 manual under the description of the Write Allocate bit (which must be set for the Amiga to run with the Data Cache).

INTUITION

- Private **IBase** variables have moved/changed. Reading them is illegal. Writing them is both illegal and dangerous.
- Poking **IBase MaxMouse** variables is now a no-op, but stop poking when Intuition version is >35.
- If you are opening on the Workbench screen, be prepared to handle larger screens, new modes, new fonts, and overscan. Also see "Font Compatibility" information.
- Screen **TopEdge** and **LeftEdge** may be negative.
- Left-Amiga-Select is used for dragging large screens. Do not use left-Amiga-key combinations for application command keys. The left-Amiga key is reserved for system use.
- For compatibility reasons, **GetScreenData()** lies if the Workbench screen is a mode only available after 1.3. It will try to return the most sensible mode that old **OpenScreen()** can open. This was necessary to prevent problems in applications that cloned the Workbench screen. To properly handle new modes, see **LockPubScreen()**, **GetVPModeID()**, and the **SA_DisplayID** tag for **OpenScreenTags()**.
- Using combined RAWKEY and VANILLAKEY now gives VANILLAKEY messages for regular keys, and RAWKEY messages for special keys (fkeys, help, etc.)
- Moving a SIMPLE_REFRESH window does not necessarily cause a REFRESHWINDOW event because layers now preserves all the bits it can.
- Sizing a SIMPLE_REFRESH window will not clear it.
- MENUVERIFY/REQVERIFY/SIZEVERIFY can time out if you take too long to **ReplyMsg()**.
- Menu-key equivalents are ignored while string gadgets are active.
- You can't type control characters into string gadgets by default. Use Ctrl-Amiga-char to type them in or use IControl Prefs to change the default behavior.
- Width and Height parameters of **AutoRequest** are ignored.
- New default colors, new gadget images.
- JAM1 rendering/text in border may be invisible gadgets over default colors.
- The cursor for string gadgets can no longer reside outside the cleared container area. If your gadget is 32 pixels wide, with MaxChars of 4, all 32 pixels will be cleared, instead of just 24, as was true in 1.3.
- Applications and requesters that fail to specify desired fonts will get the fonts the user sets up in Font Preferences in Release 2. These could be much larger, or proportional in some cases. Screen and window titlebars (and their gadgets) will be taller when accommodating a larger font. Applications which open on the Workbench screen must adapt to variable size titlebars. Any application which accepts system defaults for its screen, window, menu, **Text** or **IntuiText** fonts must adapt to different fonts and titlebar sizes. String gadgets whose height is too small for a font will revert to a smaller ROM font. There are now 2 different user-specifiable default system fonts which affect different Intuition features. This can lead to mismatches in mixed gadgets and text. See the "Intuition Screens" chapter.

- Don't modify gadgets directly without first removing them from the gadget list, unless you are using a system function designed for that purpose, such as **NewModifyProp()** or **SetGadgetAttrs()**.

- Don't rely on **NewModifyProp()** to fully refresh your prop gadget after you've changed values in the structure. **NewModifyProp()** will only correctly refresh changes which were passed to it as parameters. Use **Remove/Add/RefreshGList()** for other kinds of changes.

- Custom screens must be of type CUSTOMSCREEN or PUBLICSCREEN. Other types are illegal. One application opens its screen with **NewScreen.Type** = 0 (instead of CUSTOMSCREEN, 0x0F). Then, when it opens its windows, it specifies **NewWindow.Type** of 0 and **NewWindow.Screen** of NULL, instead of **Type**=CUSTOMSCREEN and **Screen**=(their screen). That happened to work before, but no longer.

- Referencing **IntuiMessage->IAddress** as a **Gadget** pointer on non-Gadget IDCMP messages, or as a **Window** pointer (rather than looking at the proper field **IntuiMessage->IDCMPWindow**) may now cause *Enforcer* hits or crashes. The **IAddress** field always used to contain a pointer of some type even for IDCMP events for which no **IAddress** value is documented. Now, for some IDCMP events, **IAddress** may contain a non-address, possibly an odd value that would crash a 68000 based system).

- Using Intuition flags in the wrong structure fields (for example, using ACTIVEWINDOW instead of ACTIVATE). To alleviate this problem, 2.0 has introduced new synonyms that are less confusing than the old ones. For example, IDCMP_ACTIVEWINDOW and WFLG_ACTIVATE. This particular example of confusion (there are several) was the nastiest, since IDCMP_ACTIVEWINDOW, when stuffed into **NewWindow.Flags**, corresponds numerically to WFLG_NW_EXTENDED, which informs Intuition that the **NewWindow** structure is immediately followed by a **TagList**, which isn't there! Intuition does some validation on the tag-list pointer, in order to partially compensate. To make your compiler use the new synonyms only, add this line to your code before Intuition include files: #define INTUI_V36_NAMES_ONLY.

- Do not place spaces into the **StringInfo->Buffer** of a LONGINT string gadget. Under 1.3, this worked, but the 2.0 validation routine that checks for illegal keystrokes looks at the contents for illegal (i.e., non-numeric) characters, and if any are found assumes that the user typed an illegal keystroke. The user's only options may be shift-delete or Amiga-X. Use the correct justification instead.

- If you specify NULL for a font in an IntuiText, don't assume you'll get Topaz 8. Either explicitly supply the font you you need or be prepared to size accordingly. Otherwise, your rendering will be wrong, and the user will have to reset his Preferences just to make your software work right.

- Window borders are now drawn in the screen's **DetailPen** and **BlockPen** rather than the **Window**'s pens. For best appearance, you should pass an **SA_Pens** array to **OpenScreen()**. This can be done in a backwards compatible manner with the **ExtNewScreen** structure and the NS_EXTENDED flag.

- The system now renders into the full width of window borders, although the widths themselves are unchanged. Window borders are filled upon activation and inactivation.

- Window border rendering has changed significantly for 2.0. Note that the border dimensions are unchanged from 1.x (Look at **window->BorderLeft/Top/Width/Height** if you don't believe us!). If your gadget intersects the border area, although it may have looked OK under 1.3, a visual conflict may occur under 2.0. If Intuition notices a gadget which is substantially in the border but not declared as such, it treats it as though it were (this is called "bordersniffing"). Never rely on Intuition to sniff these out for you; always declare them explicitly (see the Gadget Activation flags GACT_RIGHTBORDER, etc.). See "Intuition Gadgets and Window Borders" in the "Additional Information" section below.

PREFERENCES

- Some old struct **Preferences** fields are now ignored by *SetPrefs* (for example **FontHeight**). *SetPrefs* also stops listening to the pointer fields as soon as a new-style pointer is passed to Intuition (new-style pointers can be taller or deeper).
- **Preferences ViewX/YOffset** only applies to the default mode. You cannot use these fields to move the position of all modes.
- The **Preferences** LACEWB bit is not necessarily correct when Workbench is in a new display mode (akin to **GetScreenData()**).

WORKBENCH

- The Workbench GUI now has new screen sizes, screen top/left offsets, depths, modes, and fonts.
- Default Tool now searches paths.
- New Look (boxed) icons take more space.
- Do not use icons which have more 1bits set in **PlanePick** than planes in the **ImageData** (one IFF-to-Icon utility does this). Such icons will appear trashed on deeper Workbenches.
- New Look colors have black and white swapped (as compared to 1.3).
- The Workbench screen may not be open at startup-sequence time until some output occurs to the initial Shell window. This can break startup-sequence-started games that think they can steal WB's screen bitplanes. Do not steal the Workbench screen's bitplanes. (For compatibility, booting off pre-2.0 disks forces the initial screen open. This is not guaranteed to remain in the system.) Use startup code that can detach when RUN (such as *cback.o*) and use **CloseWorkbench()** to regain the screen's memory. See ''Workbench and Startup'' in the ''Additional Information'' section below.

LAYERS

- Use **NewLayerInfo()** to create, not **FattenLayerInfo()**, **ThinLayerInfo()**, **InitLayers()**.
- Simple-refresh preserves all of the pixels it can. Sizing a SIMPLE_REFRESH window no longer clears the whole window.
- Speed of layer operations is different. Don't depend on layer operations to finish before or after other asynchronous actions.

GRAPHICS

- Do not rely on the order of Copper list instructions. The Release 2 **MrgCop()** function builds different Copper lists to that of 1.3, by including new registers in the list (e.g., MOVE xxxx,DIWHIGH). This changes the positions of the other instructions. We know of one game that 'assumes' the BPLxPTRs would be at a certain offset in the Copper list, and that is now broken on machines running 2.0 with the new Denise chip.

- Graphics and layers functions which use the blitter generally return after *starting* the final blit. If you are mixing graphics rendering calls and processor access of the same memory, you must **WaitBlit()** before touching (or deallocating) the source or destination memory with the processor. For example, the **Text()** function is faster in Release 2, causing some programs to trash partial lines of text.

- **ColorMap** structure is bigger. Programs must use **GetColorMap()** to create one.

- Blitter rtns decide ascend/descend on 1st plane only.

- Changing the display mode of an existing screen or viewport while open is still not a supported operation.

- **GfxBase DisplayFlags** and row/cols may not match Workbench screen.

- Do not hardcode modulo values - use **BitMap->BytesPerRow**.

- If the graphics Autodocs say that you need a **TmpRas** of a certain size for some functions, then you must make that the minimum size. In some cases, before 2.0, you may have gotten away with using a smaller **TmpRas** with some functions (for example **Flood()**). To be more robust, graphics now checks the **TmpRas** size and will fail the function call if the **TmpRas** is too small.

- ECS chips under 2.0 generate displays differently. The display window registers now control DMA.

- **LoadRGB4()** used to poke colors into the active copperlist with no protection against deallocation of that copperlist while it was being poked. Under 2.0, semaphore protection of the copperlist was added to **LoadRGB4()** which makes it totally incorrect and extremely dangerous to call **LoadRGB4()** during an interrupt. The general symptom of this problem is that a system deadlock can be caused by dragging one screen up and down while another is cycling. Color cycling should be performed from within a task, not an interrupt. In general, the only functions which may be safely called from within an interrupt are the small list of Exec functions documented in the ''Exec Interrupts'' chapter.

FONTS

- Some font format changes (old format supported).

- Private format of .font files has changed (use *FixFonts* to create).

- Default fonts may be larger, proportional.

- Topaz is now sans-serif.

- Any size font will be created via scaling as long as **TextAttr.Flags** FPF_DESIGNED bit is not set. If you were asking for an extreme size, like size 1 to get smallest available, or 999 to get largest available, you will get a big (or very very small) surprise now.

- Do not use -1 for **TextAttr.Flags** or **Styles**, nor as the flags for **AvailFonts**(one high bit now causes **AvailFonts** to return different structures). Only set what you know you want. A kludge has been added to the OS to protect applications which currently pass -1 for **AvailFonts** flags.

CLI / SHELL

- Many more commands are now built-in (no longer in C:). This can break installation scripts that copy C:commandname, and programs that try to **Lock()** or **Open()** C:commandname to check for the command's existence.

- The limit of 20 CLI processes is gone and the **DOSBase** CLI table has changed to accommodate this. Under V36 and higher, you should use new 2.0 functions rather than accessing the CLI table directly.

- Shell windows now have close gadgets. The EOF character is passed for the close gadget of a Shell. This is -1L with CON: **getchar()**, and the Close Gadget raw event ESC seq with RAW:.

- Shells now use the simple-refresh character-mapped console (see "Console" notes).

- By default, CON: now opens SIMPLE_REFRESH windows using the V36/V37 console character mapped mode. Because of some differences between character mapped consoles, and SMART_REFRESH non-mapped consoles, this may cause incompatibilities with some applications. For example, the Amiga private sequences to set left/top offset, and set line/page length behave differently in character mapped console windows. The only known work-around is to recompile asking for a CON: (or RAW:) window using the SMART flag.

- Simple refresh/character mapped console windows now support highlighting and copying text with the mouse. This feature, as well as pasting text should be transparent to programs which use CON: for console input, and output. Pasted text will appear in your input stream as if the user had typed it.

- While *CONCLIP* (see s:startup-sequence) is running, programs may receive "<CSI>0 v" in their input stream indicating the user wants to paste text from the clipboard. This shouldn't cause any problems for programs which parse correctly (however we know that it does; the most common problems are outputting the sequence, or confusing it with another sequence like that for FKEY 1 which is "<CSI>0~").

- The console.device now renders a ghosted cursor in inactive console windows (both SMART_REFRESH, and SIMPLE_REFRESH with character maps). Therefore, rendering over the console's cursor with graphics.library calls can trash the cursor; if you must do this, first turn off the cursor.

- Some degree of unofficial support has been put in for programs which use SMART_REFRESH console windows, and use graphics.library calls mixed with console.device sequences to scroll, draw text, clear, etc. This is not supported in SIMPLE_REFRESH windows with character maps, and is strongly discouraged in all cases.

- Closing an Intuition window before closing the attached console.device will now crash or hang the machine.

- Under 1.2 and 1.3, vacated portions of a console window (e.g., areas vacated because of a clear, or a scroll) were filled in with the character cell color. As of V36 this is no longer true, vacated areas are filled in with the global background color which can be set using the SGR sequence "<ESC>[>##m" where ## is a value between 0-7. In order to set the background color under Release 2, send the SGR to set background color, and a form feed to clear the screen.

- Note that SIMPLE_REFRESH character mapped consoles are immediately redrawn with the global background color when changed—this is not possible with SMART_REFRESH windows.

Additional Information

TASK SWITCHING

The 1.3 Kickstart contained two task switching bugs. After an interrupt, a task could lose the CPU to another equal priority task, even if the first task's time was not up. The second bug allowed a task whose time was up to hold on to the CPU either forever, or until a higher priority task was scheduled. Two busy-waiting tasks at high priority would never share the CPU. Because the input.device runs at priority 20, usually the effect of these bugs was masked out for low priority tasks. The *ExecBase->Quantum* field had little effect because of the bugs.

For 2.0, a task runs until either its **Quantum** is up, or a higher priority task preempts it. When the **Quantum** time is up, the task will now lose the CPU. The **Quantum** was set to 16/60 second for 1.3, and 4/60 second for 2.0.

In general, the 2.0 change makes the system more efficient by eliminating unnecessary task switches on interrupt-busy systems (for example, during serial input). However, the change has caused problems for some programs that use two tasks of equal priority, one busy-waiting and one **Wait()**ing on events such as serial input. Previously, each incoming serial character interrupt would cause task context switch allowing the event-handling task to run immediately. Under 2.0 the two tasks share the processor fairly.

INTUITION GADGETS AND WINDOW BORDERS

If 2.0 Intuition finds a gadget whose hit area (gadget Left/Top/ Width/Height) is substantially inside the border, it will be treated as though it was declared in the border. This is called ''bordersniffing''. Gadgets declared as being in the border or detected by Intuition as being in the border are refreshed each time after the border is refreshed, and thus aren't clobbered.

Noteworthy special cases:

1) A gadget that has several pixels not in the border is not bordersniffed. An example would be an 18-pixel high gadget in the bottom border of a SIZEBBOTTOM window. About half the gadget will be clobbered by the border rendering.

2) A gadget that is not substantially in the border but has imagery that extends into the border cannot be sniffed out by Intuition.

3) A gadget that is substantially in the border but has imagery that extends into the main part of the window will be sniffed out as a border gadget, and this could change the refreshing results. A common trick to put imagery in a window is to put a 1x1 or 0x0 dummy gadget at window location (0,0) and attach the window imagery to it. To support this, Intuition will never bordersniff gadgets of size 1x1 or smaller.

   All these cases can be fixed by setting the appropriate GACT_xxxBORDER gadget Activation flag.

4) In rare cases, buttons rendered with **Border** structures and JAM1 text may appear invisible under Release 2.

The height of the window's title bar is affected by the current font settings. See the discussion of "Screen Attributes" in the "Intuition Screens" chapter. To predict your window's titlebar height before you call **OpenWindow()**:

```
topborder = screen->WBorTop + screen->Font->ta_YSize + 1
```

The screen's font may not legally be changed after a screen is opened.

Be sure the screen cannot go away on you. This is true if:

1) You opened the screen yourself.

2) You currently have a window open on the screen.

3) You currently hold a lock on this screen (see **LockPubScreen()**).

IntuiText rendered into a window (either through **PrintIText()** or as a gadget's **GadgetText**) defaults to the Window **RastPort** font, but can be overridden using its **ITextFont** field. Text rendered with the **Text()** function appears in the Window **RastPort** font.

The Window's **RPort**'s font shown above is the initial font that Intuition sets for you in your window's **RastPort**. It is legal to change that subsequently with **SetFont()**.

WORKBENCH AND STARTUP

Under 1.3, the Workbench Screen and initial Shell (CLI) opened before the first line in s:startup-sequence. Some naughty programmers, in an attempt to recover memory, would search for the bitplane pointers and appropriate the memory for their own use. This behavior is unsafe.

By default 2.0 opens the initial CLI on the first _output_ from the s:startup-sequence. This allows screen modes and other parameters to be set before the user sees the screen. However, this broke so many programs that we put in the "silent-startup" hack. A disk installed with 1.3 install opens the screen as before. A disk installed under 2.0 opens silently. Never steal the Workbench bitplanes. You don't know where they are, how big they are, what format they may be in, or even if they are allocated. Recovering the memory is a bit tricky.

Under 2.0, simply avoid any output from your s:startup-sequence. If your program opens a screen it will be the first screen the user ever sees. Note that if ENDCLI is ever hit, the screen will pop open.

Under 1.3, after ENDCLI, use **CloseWorkbench()** to close the screen. This also works under 2.0. Loop on **CloseWorkbench()** with a delay between loops. Continue looping until **CloseWorkbench()** succeeds or too much time has passed. Note that a new program called *EndRun* is available for starting non-returning programs from the startup-sequence. *EndRun* will reduce memory fragmentation and will close Workbench if it is open. *EndRun.lzh* will be available in Commodore's Amiga listings area on BIX.

INDEX

Amiga Programming

>$38.95 USA
>$50.95 CANADA

THIRD EDITION

AMIGA TECHNICAL REFERENCE SERIES

AMIGA® ROM Kernel Reference Manual

LIBRARIES

The Amiga computers are exciting high-performance microcomputers with superb graphics, sound, multiwindow and multitasking capabilities. Their technologically advanced hardware is designed around the Motorola 68000 microprocessor family and sophisticated custom chips. The Amiga's unique system software provides programmers with unparalleled power, flexibility, and convenience in designing and creating programs.

Written by the technical experts at Commodore-Amiga, Inc. who design the Amiga hardware and system software, the ***Amiga® ROM Kernel Reference Manual: Libraries*** presents tutorials and detailed examples showing how to get the most from the hundreds of functions in the Amiga's system libraries. This new edition has been completely revised and updated for Release 2, the most current version of the Amiga's operating system. It includes:

- A tutorial and comprehensive reference on how to program the Amiga's system libraries including the multitasking Exec, and Intuition, the Amiga's graphical user interface
- Complete coverage of the new libraries in Release 2 including: *gadtools, iffparse, asl, keymap, utilities, workbench,* and *commodities*
- Dozens of working example programs illustrating the techniques used to create Amiga application software

For the serious programmer who wants to take full advantage of the Amiga's state-of-the-art capabilities, the ***Amiga ROM Kernel Reference Manual: Libraries*** is a vital source of information on how to use the latest system software in the whole family of Amiga computers.

The AMIGA TECHNICAL REFERENCE SERIES has been revised and updated to provide a comprehensive reference and tutorial for the entire line of Amiga computers and for Release 2 of the operating system. Other titles in the series include:

Amiga User Interface Style Guide
Amiga ROM Kernel Reference Manual: Includes and Autodocs, Third Edition
Amiga ROM Kernel Reference Manual: Devices, Third Edition
Amiga ROM Kernel Hardware Reference Manual, Third Edition

ISBN 0-201-56774-1
56774

Cover design by Hannus Design Associates

Addison-Wesley Publishing Company, Inc.